Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at
TU WIEN UB Universitätsbibliothek
The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

DISSERTATION

Interactive Model-Driven Generation of Graphical User Interfaces for Multiple Devices

Submitted at the
Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Doktor der technischen Wissenschaften

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann KAINDL
Institute of Computer Technology
Vienna University of Technology

and

Univ.Prof. Dr.sc.inf. Jean VANDERDONCKT
Louvain School of Management
Université Catholique de Louvain

and

Dipl.-Ing. Dr.techn. Roman POPP
Institute of Computer Technology
Vienna University of Technology

by

David Raneburger
Matr.Nr. 0125896
Hellwagstrasse 2/20, 1200, Wien

24.06.2014

Kurzfassung

Die Entwicklung von Graphischen User Interfaces (GUIs) ist zeitaufwändig und teuer, insbesondere wenn für dieselbe Applikation GUIs für verschiedene Endgeräte (z. B., Smartphone, Tablet oder Desktop PC) benötigt werden. Modell-getriebene GUI-Generierung erlaubt es potentiell den Entwicklungsaufwand durch automatische Transformationen von Geräte-unabhängigen Interaktionsmodellen in GUI-Quellcode zu verringern. Allerdings ist die Benutzbarkeit von automatisch generierten GUIs meist nicht zufriedenstellend. Manuelle Anpassungen erlauben es, die Benutzbarkeit zu verbessern, werden aber im Falle einer Neugenerierung üblicherweise nicht automatisch mitberücksichtigt.

Diese Arbeit erweitert einen Modell-basierten GUI Generierungsansatz – die Unified Communication Platform (UCP) – um zwei Aspekte. Erstens sollte die automatische Transformation von Interaktionsmodellen in Geräte-spezifische GUIs besser unterstützt werden und zweitens sollte der Designer mehr Kontrolle über die resultierenden GUIs bekommen, um gewünschte Anpassungen schon während des Generierungsprozesses vornehmen zu können. Die automatische Anpassung des GUIs an ein bestimmtes Endgerät durch Optimierung sollte dessen Benutzbarkeit verbessern. Solch eine Anpassung erlaubt es, die Benutzbarkeit von GUIs für Geräte mit kleinen Displays (z. B. Smartphones) zu erhöhen, da diese durch ihre geringe Displaygröße stärker eingeschränkt sind als Geräte mit großen Displays (z. B. Desktop PCs). Im Allgemeinen wird solch eine automatische Anpassung nicht ausreichen, um die gewünschte Benutzbarkeit bzw. den gewünschten „Look & Feel “zu erreichen. Wir präsentieren deshalb einen interaktiven GUI-Generierungsansatz, der es dem Designer erlaubt, zwischen unterschiedlichen Anpassungsstrategien zu wählen und das resultierende GUI durch spezielle Transformationsregeln und manuelle Adaptierungen eines „Screen-basierten “GUI Models anzupassen. Die UCP-basierte Implementierung unseres Generierungsansatzes unterstützt die vollautomatische Generierung des GUI-Quellcodes und von Teilen der Applikationslogik. Basierend auf diesem Tool präsentieren wir einen iterativen und inkrementellen Prozess für Interaktionsdesign und GUI-Anpassungen, der es erlaubt, GUIs mit guter Benutzbarkeit zu entwickeln, sowohl deren Verhalten als auch deren Struktur betreffend.

In einer früheren Version von UCP wurde bereits die automatische Transformation von Geräte-unabhängigen Interaktionsmodellen, sogenannten Diskurs-basierten Kommunikationsmodellen, zu Web-basierten GUIs unterstützt. UCP wurde in dieser Arbeit um ein Screen-basiertes Modell (das sogenannte Screen Modell) erweitert. Dieses Screen-basierte Modell definiert die Struktur der einzelnen GUI-Schirme. Es bildet die Basis sowohl für unsere automatische Geräteanpassung als auch für manuelle Anpassungen. Wir benutzten unseren Transformationsansatz und unseren iterativen und inkrementellen Entwicklungsprozess, welcher ebenfalls UCP nutzt, zur Entwicklung von zwei kleineren Applikationen (Flugbuchung und Fahrradverleih) und einer etwas größeren Urlaubsplanungsapplikation. Mittels dieser drei Applikationen zeigen wir die prinzipielle Umsetzbarkeit unseres konzeptionellen Ansatzes. Abschließend präsentieren wir eine Evaluierung der damit entwickelten GUIs.

Abstract

Graphical User Interface (GUI) development is time-consuming and error-prone, especially if GUIs for different devices (e.g., smartphone, tablet or desktop PC) are needed for the same application. Model-driven GUI generation approaches have the potential to decrease the development effort through applying automated transformations while refining device-independent high-level interaction models to GUI source code. However, the usability of such automatically generated GUIs is typically rather low. Manual customizations on GUI models can be applied to improve the usability, but are typically lost in case of regeneration.

This work extends a previous model-driven GUI generation approach – the Unified Communication Platform (UCP) – to support the automated transformation of device-independent high-level interaction models to device-tailored GUIs, and to give the designer more control over the resulting GUIs. Tailoring a GUI, through optimizing it for a specific device, improves its usability. This helps to achieve a good level of usability for devices with a small screen (e.g., smartphones), because they are more constrained in terms of screen size than devices with a large screen (e.g., desktop PC). In general, however, even an optimized GUI may still not be sufficient to achieve the desired level of usability or the desired "look & feel" of a GUI. We, therefore, propose an *interactive* GUI generation approach that allows the designer to select between different tailoring strategies and to customize the optimized GUI through specific transformation rules or adaptations on a screen-based GUI model. The UCP-based implementation of our conceptual approach supports the fully automatic generation of the GUI code and parts of the application back-end. Based on this tool, we defined an iterative and incremental process for interaction design and GUI customization, to achieve a good level of usability for the GUI's behavior and its structure.

In its previous version, UCP already supported the transformation of device-independent high-level interaction models, so-called Discourse-based Communication Models, to Web-based GUIs. We extended UCP to provide a screen-based GUI model (the Screen Model). The Screen Model provides the basis for both, our automated approach for tailoring the GUI for a specific device, and for manual customization. We applied our new approach and our iterative and incremental development process using UCP, to develop two small applications (flight booking and bike rental), and a more elaborate vacation booking application. This allowed us to test the feasibility of our conceptual approach and to evaluate the usability of the generated GUIs.

Acknowledgements

Das Schreiben einer Dissertation kostet eine gewisse Menge an Zeit und Nerven und wäre ohne Unterstützung aus dem privaten und dem fachlichen Umfeld vermutlich nur sehr schwer bzw. nicht in angemessener Qualität machbar.

Ich möchte mich hier an erster Stelle bei meiner Frau Julia für die Geduld und Motivation bedanken, ohne die ich diese Dissertation vermutlich nie fertig geschrieben hätte. Ein großes Dankeschön für moralische Unterstützung geht auch an alle Mitglieder meiner Familie!

Bei meinem Betreuer Professor Hermann Kaindl möchte ich mich sehr herzlich für die vielen fachlichen Diskussionen, Korrekturen und Anmerkungen bedanken, ohne die diese Dissertation vermutlich sehr schwer lesbar geblieben wäre.

I want to express my appreciation to Professor Vanderdonckt for supervising me all the years, since we first met at the Doctoral Consortium at EICS'10 in Berlin. Thank you very much for always having an open ear for my questions, your scientific input, for coming all the way from Brussels to my defense and for being such an inspiring scientist!

Für eine hervorragende fachliche Betreuung und viele hilfreiche Diskussionen und Hilfestellungen möchte ich mich sehr herzlich bei meinem Betreuer Dr. Roman Popp bedanken.

Ein großer Dank geht auch an Barbara Weixelbaumer, Astrid Weiß, Nicole Mirnig, Verena Fuchsbaumer und Brigitte Ratzer für die lehrreiche und angenehme Zusammenarbeit im GENUINE Projekt.

Weiters möchte ich mich bei meinen ehemaligen Arbeitskollegen Jürgen Falb, Dominik Ertl und David Alonso-Rios für viele Tipps und Hilfestellungen bedanken, die ich im Laufe der Zusammenarbeit erhalten habe und bei Michael Leitner, Alexander Schörkhuber, Vedran Sajatovic und Alexander Armbruster für die gute Zusammenarbeit.

Zum Schluss bleibt mir noch der Dank an Reinhard Hametner, Christian Panzer, Wilfried Lepuschitz und Alexander Prostejovsky. Ohne eure Freundschaft und den damit verbundenen Erfahrungsaustausch würde ich denken es hätte leichter sein können :-).

Wien, Juni 2014

Table of Contents

1 Introduction **1**
- 1.1 Motivation and Research Problem . . . 2
- 1.2 Terminology . . . 4
- 1.3 Proposed Solution and Research Hypotheses . . . 5
- 1.4 Outline . . . 8

2 Background and State-of-the-Art **11**
- 2.1 Conceptual Background on Model-driven UI Generation for Multiple Devices . . . 11
- 2.2 State-of-the-Art UI Generation Approaches . . . 16
- 2.3 The Unified Communication Platform . . . 24

3 Automated Screen Model Generation **34**
- 3.1 WIMP-UI Behavior Generation . . . 35
- 3.2 Screen Model Generation . . . 41

4 (Semi-)Automatic GUI Tailoring for Multiple Devices **47**
- 4.1 Application Tailored Device Specification . . . 48
- 4.2 Transformation Rules for Device Tailoring . . . 48
- 4.3 Automated Layout Calculation . . . 55
- 4.4 Automated Device Tailoring . . . 69

5 Interaction Model Development and GUI Customization for Multiple Devices **86**
- 5.1 Iterative Interaction Model Development . . . 88
- 5.2 Iterative and Incremental Interaction Model Development . . . 96
- 5.3 Customization through Custom Transformation Rules . . . 98
- 5.4 Customization through Screen Model Adaptations . . . 102
- 5.5 Customization for Multiple Devices . . . 108

6 Evaluation and Results **111**
- 6.1 Evaluation of our Iterative Development Process using Bike Rental . . . 111
- 6.2 Evaluation of our Iterative and Incremental Development Process with Customization for Multiple Devices using Vacation Planning . . . 116
- 6.3 Evaluation of (Semi-)automatically Generated GUIs . . . 130
- 6.4 Key Contributions and Results of this Doctoral Dissertation . . . 157
- 6.5 How the Results Support the Hypotheses . . . 158

7 Discussion **160**

8 Future Work **162**

9 Conclusion **163**

A Key Features of Model-based User Interface Generation Approaches **164**
A.1 General Features . . . 166
A.2 Input Model Features . . . 169
A.3 Method Features . . . 170
A.4 Application Features . . . 172
A.5 Tool Features . . . 174

Literature **176**

Abbreviations

ADJ	Adjacency Pair
ANM	Action-Notification Model
ATL	Atlas Transformation Language
AUI	Abstract User Interface
BP	Best Practice
BPMN	Business Process Model and Notation
CASE	Computer-aided Software Engineering
CIM	Computational Independent Model
CRF	Cameleon Reference Framework
CSS	Cascading Style Sheets
CSV	Comma Separated Value
CTT	ConcurTaskTree
CUI	Concrete User Interface
DIN	Deutsches Institut für Normung
DoD	Domain-of-Discourse
DPI	Dots per Inch
EMF	Eclipse Modeling Framework
ETS	Enabled Task Set
FUI	Final User Interface
GSME	Graphical Screen Model Editor
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IFML	Interaction Flow Modeling Language
ISM	Implementation Specific Model
ISO	International Organization of Standardization
JET	Java Emitter Templates
LHS	Left Hand Side
MBUID	Model Based User Interface Development
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MOF	Meta Object Facility
MVC	Model-View-Controller
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
PC	Personal Computer
PIM	Platform Independent Model
PSM	Platform Specific Model
PTS	Presentation Task Set

PU	Presentation Unit
QVT	Query/View/Transformation Language
RHS	Right Hand Side
RST	Rhetorical Structure Theory
SAP	State-Action Pair
SOA	Service Oriented Architecture
SQL	Structured Query Language
SUS	System Usability Scale
UCP	Unified Communication Platform
UCP:CM	Unified Communication Platform Communication Model
UCP:RT	Unified Communication Platform Run-time Framework
UCP:UI	Unified Communication Platform UI Generation Framework
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WIMP	Window/Icon/Menu/Pointing Device
WSDL	Web Service Description Language

1 Introduction

User Interface development is time-consuming and error-prone. For example, a survey on user interface programming revealed that the Graphical User Interface (GUI) of an interactive system required about 45% of the development time, 50% of the implementation time and represents about 48% of the source code. The required maintenance time is about 37% of the maintenance time for the whole system [MR92]. These figures, evaluated in the early nineties, are increasing dramatically with an increasing number of devices to access an application or ubiquitous computing environments where such devices are required to interact in an ad-hoc manner.[1]

Model-*based* user interface development provides a means to decrease the development effort through the use of high-level models that are refined for a certain context of use (i.e., device, user, environment) [MPV11]. Model-*driven* UI development does not only include models in the user interface (UI) development process, but even uses them to "drive" it. In particular, model-driven UI development puts the models at the center of the refinement process and applies automated transformations to translate high-level models to source code. These transformations typically consist of model-to-model and model-to-code transformations that refine the high-level model over different levels of abstraction, deriving intermediate models before the final source code is generated.

The usability of the resulting UIs is typically rather low if the transformations are completed fully automatically [MPV11]. A major reason is that the automated transformations need to be designed in a generic way (e.g., based on heuristics) to support different kinds of applications and thus, do not take requirements for a certain application and the device this application is running on into account.

Including the designer in the transformation process, and thus giving her the possibility to add application-specific information at an appropriate level of abstraction, remedies this problem, but increases the development effort [PHDB12]. In addition, creating adequate models requires time and effort as well, and to the best of our knowledge there is no proof so far that this is less than required by manual UI development. So, there is a trade-off between development effort and quality of the resulting UI just like in manual development.

We conjecture that the development effort for a certain UI is comparable to the manual effort required. So, the full advantages of model-driven UI development, in terms of saving development time and effort while developing an interactive application, can only be gained if such high-level interaction models allow for the generation of tailored UIs for multiple devices in an automated

[1] `http://www.w3.org/wiki/MBUI_Working_Group_Mission`

way. Furthermore, we suggest to facilitate the automated exploration of design alternatives and giving the designer the possibility to provide extra input to achieve a good level of usability, including the desired "look & feel", for the resulting UIs.

1.1 Motivation and Research Problem

Today, developers of User Interfaces face a diverse environment from both a technical and a human perspective [MCC13]. Technical diversity includes a large number of different programming/markup languages and computing devices. Human diversity includes the human user as a person and her different needs according to the working environment she is currently facing (e.g., a desktop workstation in an office vs. operating a smartphone in a train).

Model-based UI development provides a means to cope with this diversity through the use of high-level models that capture only aspects that are independent of a certain technical and/or human perspective. Such abstract models are subsequently enriched with the information that is specific for a certain perspective and thus tailored for a certain context of use (i.e., user, platform, environment). Model-based UI development primarily exploits the communicative value of models, using them to foster mutual understanding between experts of different domains, which makes them an optional add-on in the development process.

Model-driven software development also relies on the concept of using higher-level models, but additionally introduces automated transformations to refine these models to source code, with the objective to increase productivity and reduce time-to-market, compared to programming languages [SK03]. Model-driven UI generation is desirable, as it potentially saves development time and effort, promising that "when the user's requirements or the context of use change, the models change accordingly and so does the supporting user interface [Van08]". Model-driven UI generation is especially promising in today's environments where new technologies emerge rapidly and the same Websites or applications are typically accessed with different devices (e.g., smartphone, tablet or desktop PC), where each device may apply a different computing platform (i.e., operating system and UI toolkit/markup language). This plethora of devices increases the UI development effort, as UIs typically need to consider device-properties (e.g., display size or mouse vs. touch operation) in addition to platform characteristics to achieve a good level of usability.

One of the key challenges is to enrich device-independent high-level models with information during the generation process to be finally able to generate device-specific source code [SK03]. The Object Management Group's (OMG) Model-driven Architecture (MDA)[2] specifies a conceptual approach to automatically transform high-level models to an implementation (i.e., executable source code). A few approaches for model-driven software development have been reported that already applied MDA successfully in industrial settings[3], and there is also a small number of approaches that support fully-automatic UI generation from high-level interaction models. These approaches have rarely been applied in industrial settings so far, as the usability of the resulting UIs is typically rather low [MPV11].

One way to address this problem are *semi*-automatic UI development approaches that are *interactive* by keeping the designer in the loop during the development process. *Inter-action* literally

[2] `http://www.omg.org/mda/`
[3] `http://www.omg.org/mda/products_success.htm`

means that there is *action between* two actors, in the context of this thesis between the designer and a model-driven UI generation process. This definition allows either actor to initiate the interaction, and does not explicitly require a certain actor to start it. According to this definition even automatic UI generation, where human intervention (i.e., interaction) is potentially supported but not compulsory, is interactive. We claim that an interactive UI development approach should support fully-automated UI generation to achieve a low development effort. As software development in general, and UI development in particular, include the selection of design alternatives, such an approach should be configurable to facilitate the exploration of alternatives.

Support for the exploration of design alternatives is beneficial as it allows for their early comparison, testing and detection of failing options. However, we conjecture that application-specific input through the designer will always be required to achieve a good level of usability. Support for interaction in the context of model-driven UI generation should allow the designer to provide additional input for the generation process on different levels of abstraction, through modifying the respective models and/or transformations. Such additional input can be, e.g., style or layout specifications that allow the designer to achieve the desired *"look & feel"*, required for a good level of usability.

Additional input is hardly ever available at a certain point in time, but rather the result of exploring alternatives through manual adaptations in various iterations. In the context of UI development, such iterative refinement is crucial for achieving high-quality UIs (i.e, UIs with a good level of usability). Hence, it is not only necessary to include the designer in the development cycle and to facilitate the exploration of design alternatives, but also to persist her modifications in case of re-generation, which is typically not supported in current state-of-the-art UI generation approaches [MPV11].

This work focuses on the model-driven generation of Graphical User Interfaces (GUIs), more precisely on Window / Icon / Menu / Pointing Device (WIMP) GUIs, at design-time. We propose to tackle the basic problem of how to achieve a high level of usability for generated GUIs while still getting the benefits of model-driven development, in two steps. We formulate these steps as two *research problems*:

1. *Support for fully-automated generation of different device-specific GUIs* to facilitate the exploration of alternatives and to achieve a basic level of usability without any extra effort.
2. *Support for iterative manual GUI adaptations through the designer*, giving her the ability to further improve the usability of the resulting GUI.

Research problem 1 is typically referred to as multi-device GUI generation with varying degrees of automation in state-of-the-art literature [MPS04, PRK13]. These multi-device GUI generation approaches allow for (semi-)automatic tailoring for a certain device, which has been identified as an important ingredient to achieve a good level of usability [AVCF+10], but do not support the automated generation of device-specific alternatives. Alternatives have to be explored manually.

Research problem 2 is referred to as the beautification problem in state-of-the-art literature [PVE+07, AVVP09], where manual adaptations are referred to as customizations. However, there is no support for iterative customization as current state-of-the-art approaches typically lose all customizations in case of regeneration [MPV11].

Figure 1.1 summarizes the basic problem of how to transform device-independent high-level interaction models automatically to device-dependent GUIs. For example, smartphones and tablet

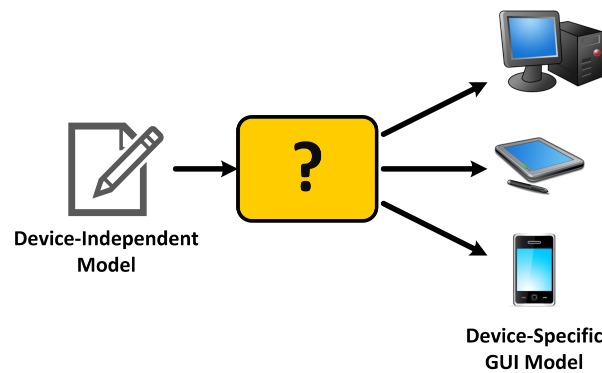


Figure 1.1: Basic Problem: How to transform the same device-independent model to different device-dependent models?

PCs are touch-based and typically require GUIs with different widgets or a different layout than desktop PCs, where usually a mouse and a keyboard are used for interaction. Moreover, all these devices differ in the amount of available space on a specific display (i.e., display size). Device-dependent GUI models are tailored for a certain device (i.e., they consider device properties like available display size) and we therefore, consider the notions device-dependent, device-specific and device-tailored as synonyms in the context of this thesis.

1.2 Terminology

Before proposing a solution for the research problems formulated above we clearly define the meaning of five terms in the context of this doctoral dissertation. This is important to achieve a precise formulation of our hypotheses in the next section.

Device

We rely here on the device definition published in the glossary of the W3C working group for Model-based UI (MBUI) development:

> "An apparatus incorporating, or managing, a collection of peripherals, and which appears to a user as a functional unit through which to perform an interaction process. A device can include computational abilities, act as a stand-alone interactive system, or be part of a network. Examples include iPhone 4, Samsung Galaxy III, MS Kinect, Wii...".[4]

Display

A display is the physical display of a specific device. One of its properties is its *display size*, which is given through a display width, a display height and dots per inch.

[4] http://www.w3.org/TR/mbui-glossary/#device_def

Platform

This work relies on the platform definition of the Cameleon Reference Framework, which considers software and hardware properties explicitly [CCT+03]. This allows us to use device and platform as synonyms. The term platform can also be interpreted differently based on the platform definition of the Model Driven Architecture [MM03]. The correspondence between these two definitions is discussed in Section 2.1.

Multi-Device User Interface (Multi-device UI)

Multi-device user interfaces allow the user to access a given application with multiple devices (e.g., desktop PC or smartphone). This work focuses on graphical user interfaces for multiple devices, which provide the same functionality, but may distribute the widgets to different screens (e.g., through splitting an existing screen), they may use different widgets, different layout or style. This definition differs from cross-platform or cross-device UIs. These two notions are used in the context of migratory and distributed UIs, which are both out of scope for this work. Our definition is compliant with the multi-device definition in the W3C MBUI glossary:

> "Multi-device user interfaces can be accessed through multiple devices."[5]

, but further constrains it, as we focus on GUIs and put migratory and distributed UIs out of scope.

Screen

A screen is a container for concrete interaction widgets (e.g., labels or buttons) that are concurrently available at a certain point in time. In principle, a screen can be on different levels of abstraction. For example, it can be represented through source code (e.g., a Web-page), or it can be represented through a toolkit-independent GUI model (i.e., on the Concrete UI Level of the Cameleon Reference Framework [CCT+03]). The concept can actually also be mapped to high-level interaction models, where a screen can be represented, e.g., through a Presentation Task Set [PBSK99] or a Presentation Unit [RPKF11]. A GUI typically consists of several screens.

Usability

This work relies on the usability definition provided in ISO 9241-11:

> The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

1.3 Proposed Solution and Research Hypotheses

We propose to extend established *model-driven UI generation concepts* so that they support *automated device-tailoring* and allow for *iterative human intervention at an appropriate level of*

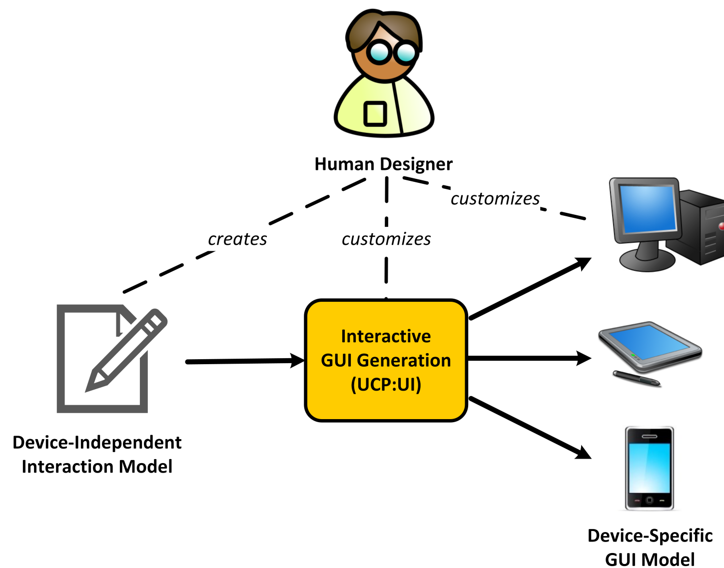


Figure 1.2: Proposed Solution: Interactive Multi-Device GUI Generation (UCP:UI)

abstraction. In particular, we focus on Graphical User Interfaces (GUIs) and propose to make GUI generation interactive, as illustrated in Figure 1.2.

Interactive generation of graphical user interfaces, according to our definition, includes a human designer in the generation process, requiring her to provide an adequate interaction model and giving her the possibility to customize the transformation process and the resulting device-specific GUI models, before the source code is generated. This implies that our approach transforms the models involved to source code at *design-time*, as opposed to run-time interpretation, where the designer does not have the opportunity to customize any models that are derived from the input models.

This doctoral dissertation presents our *conceptual approach* for *Interactive Multi-device GUI Generation* together with its *implementation* in the *Unified Communication Platform UI Generation Framework (UCP:UI)*. We explain in detail how automated multi-device GUI generation is achieved in UCP:UI together with an evaluation of our approach. To make our concept of "Interactive Model-Driven Generation of Graphical User Interfaces for Multiple Devices" completely transparent we made its implementation available under a free license, together with a fully implemented example, on our Website[6].

Figure 1.3 illustrates the basic conceptual approach of this doctoral dissertation. The proposed approach builds on Discourse-based Communication Models [FKH+06, Pop12] that support the specification of the communicative interaction between two parties as classes of dialogues that are device-independent. Transformation rules are applied to transform a Communication Model into a device-specific Screen Model, which specifies all screens of the application toolkit-independently. The Automated Device Tailoring is configurable and allows for generating different device-specific GUIs (optimized according to different tailoring strategies) for a given Device Specification, facilitating the exploration of design alternatives.

[5] http://www.w3.org/TR/mbui-glossary/
[6] http://ucp.ict.tuwien.ac.at

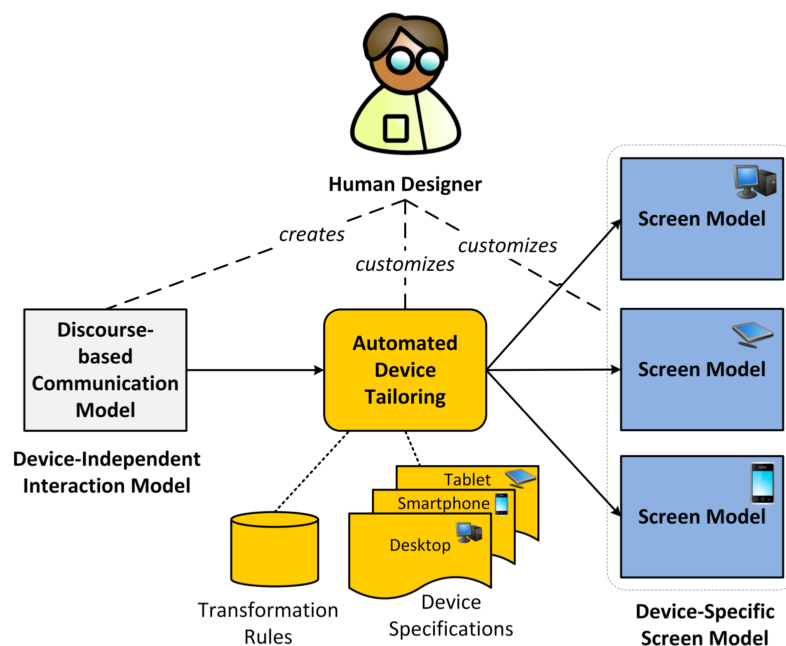


Figure 1.3: Basic Conceptual Approach for Interactive Multi-Devie GUI Generation (UCP:UI)

The proposed approach keeps the human designer in the loop and allows for customization at two different levels of abstraction. First, it provides the possibility to customize the automated tailoring process through selecting different tailoring strategies or adding application-specific transformation rules and second it provides a device-specific GUI model for manual layout and style customizations. Overall, this approach implements the reification of Discourse-based Communication Models to Source Code [CCT+03]. Other development paths, like support for reverse engineering through abstraction, are out of scope for this work.

We formulated the key ideas of this doctoral dissertation in three hypotheses[7].

Hypothesis 1: **Device-tailored GUIs provide better usability than non-tailored GUIs.**

GUIs (e.g., Websites) have been developed and tested for a certain device in the past, typically a desktop PC where the user operates the GUI using keyboard and mouse. Today exists a variety of additional devices (e.g., smartphones and tablet PCs) that do not only differ from desktop PCs regarding their display size and computational power, but also through supported toolkits and their touch-based operation. In addition, there are different ways to tailor GUIs for a certain device that should be compared to identify the most appropriate one. We conjecture that GUIs that take such device characteristics into account and are tailored to a certain device in an appropriate way, provide better usability than GUIs that do not take such characteristics into account.

[7] The key ideas have been formulated in two hypotheses in [Ran10]. The first hypothesis: *Interactive model driven UI generation results in more satisfying UIs* was replaced in this doctoral dissertation through two more concrete ones, presented in the following as Hypothesis 1 and 2. The second hypothesis of [Ran10] was reformulated to be more precisely and is consequently presented as Hypothesis 3.

Hypothesis 2: **Manually customized GUIs provide better usability than fully-automatically generated GUIs.**

Layout and style information is typically added based on heuristics in model-driven GUI generation approaches. However, it is vital to consider application-specific style and layout requirements to achieve a high level of usability for the resulting application. This information can typically not be specified in a high-level interaction model and has to be added either in the form of annotations of the high-level models, on the lower levels of abstraction, or through specific transformation rules. A model-driven GUI generation approach that aims for high-quality GUIs needs to involve the designer in the generation process and to provide access to its models and artifacts on an appropriate level of abstraction (e.g., GUI customization should be supported on Concrete UI Level of the Cameleon Reference Framework [CCT+03]).

Hypothesis 3: **Interactive Multi-Device GUI Generation (UCP:UI) supports iterative GUI development.**

Iterative development is a crucial ingredient to achieve high quality UIs [PRS11]. It is especially useful in a GUI development process as it allows for early involvement of end-users. To support iterative development in a model-driven context, such approaches need to persist all customizations and re-apply them in case of re-generation.

This doctoral dissertation tackles the two *research problems* formulated above through extending the Unified Communication Platform UI Generation Framework UCP:UI[8] with the *Screen Model*, a screen-based representation of a device-specific GUI. We show that the Screen Model both can be used to automatically tailor the resulting GUI to a certain device in different ways, and that it provides the basis for manual style and layout customizations on an appropriate level of abstraction. We finally test the three *hypothesis* formulated above to evaluate the presented concepts and their implementation.

Overall this work focuses on the technical perspective of dealing with the diversity of devices and platforms. Its basic concepts regarding tailoring could also be extended to include aspects of the human perspective, which is out of scope for this work.

1.4 Outline

Following this introduction, Chapter 2 presents background information for the conceptual approach and implementation presented in this doctoral dissertation, and state-of-the-art approaches UI generation approaches. It starts with presenting and comparing two established conceptual approaches for multi-device UI development, the Object Management Group's Model Driven Architecture [MM03] that supports model-driven software development in general, and the Cameleon Reference Framework [CCT+03] that allows for classifying Model-based UI development approaches. We compare and relate both conceptual approaches to identify characteristics for multi-device GUI generation and a clear definition of how their respective notions of "platform" correspond. Subsequently, we present state-of-the-art UI generation approaches that comply to the conceptual approaches mentioned above. This comparison is based on a set of UI generation approach properties that we defined as *key features*. Our key features allow for the classification

[8] `http://ucp.ict.tuwien.ac.at`

of UI generation approaches and facilitate their systematic comparison. We use a subset of our key features for our systematic comparison (the full set is defined in Appendix A) to give a short outlook on current trends and future research and motivate why we chose the Unified Communication Platform[9] to implement our new concepts. The last section in this chapter presents all details of the Unified Communication Platform that are required to understand the remainder of this doctoral dissertation.

Chapter 3 presents our interactive model-driven GUI generation process and its UCP-based implementation. It shows how we extended and combined an existing approach for the automated generation of GUI structures [Kav11] with an existing approach for the automated derivation of a UI behavior model [PFA+09] to automatically generate a screen-based GUI model – the Screen Model. In particular, our new approach automatically transforms a Discourse-based Communication model into a Structural and a Behavioral Screen Model by weaving a device-independent WIMP-UI Behavior Model and a device-specific Structural UI Model.

Chapter 4 presents our Platform Model and how the Screen Model can be used for automated multi-device GUI generation. We model a platform through a so-called Application-tailored Device Specification, which specifies hardware and software properties of a given device, based on [KRF+09]. For example, such a device specification defines how a device is used by a certain application (e.g., is scrolling allowed, and if yes up to which length). In addition, automated multi-device GUI generation requires a complete set of transformation rules, a layout engine and at least one implemented strategy for tailoring the GUI for a certain device. So, we continue with presenting our concepts and implementation for achieving a complete set of transformation rules that supports fully-automated multi-device GUI generation, followed by our configurable automated layout approach that supports the specification of layout hints through the designer. Finally, we present our approach for automated device-tailoring. In particular, we present our objective function and four different tailoring strategies that allow the designer to explore different alternatives.

Chapter 5 introduces a tool-supported process for iterative interaction model development. Such a process facilitates the exploration and evaluation of interaction design alternatives, using automated GUI generation to achieve a running prototype more quickly and with reduced effort in comparison to manual (prototype) development. This facilitates the development of high-quality interaction models, which are a pre-requisite for a high level of usability of the final application, because they define the behavior of the application. We subsequently extend this process to support incremental interaction development, which is typically applied when developing more complex applications. These require more complex interaction models, which means that also the number of possible GUIs is increasing. Next, we show how the number of possible GUIs can be reduced through application-specific transformation rules. Such transformation rules can also be used for customizing the GUI to achieve the desired "look & feel". Together with customizations on the Screen Model, which we store in a so-called "Changes Model" and re-apply in case of re-generation, we support the customization of all UI aspects identified in [PHKB12]. Finally, we show how customizations performed for a specific device can be partially reused when customizing the GUI for another device.

Chapter 6 evaluates our iterative interaction design process using a small Bike Rental application, generating a smartphone and a desktop GUI. Next, we evaluate the iterative and incremental development process and the customization through application-specific transformation rules using a more complex Vacation Planning application, again with a smartphone and a desktop GUI.

[9] `http://ucp.ict.tuwien.ac.at`

We also provide data on additional customization effort for both GUIs and usability evaluation results of the generated GUIs. To provide data on the quality of our generated GUIs in terms of usability, we will present the results of Heuristic Evaluations and user studies. In particular, we will report on a Heuristic Evaluation of two fully automatically generated GUIs (i.e., desktop and smartphone) for our small Bike Rental application. Next we will report on the results of a user study that investigated different tailoring strategies for tailoring a GUIs for small devices (e.g., smartphones) and finally we will summarize the results of a user study that was performed at the University of Salzburg, based on two customized GUIs (i.e., desktop and smartphone) for our Vacation Planning application. Eventually, we summarize the results of this doctoral dissertation and show how the results support the hypotheses.

Chapter 7 discusses interesting design decisions that have been made while developing our GUI generation approach and applying it during the development of the applications presented in the previous evaluation chapter.

Chapter 8 presents our ideas for future improvements / extensions of existing concepts and their implementation and new concepts.

Chapter 9 summarizes the presented work and concludes this doctoral dissertation.

Appendix A presents the full list of UI generation approach properties that we defined as key features. These key features support the classification of UI generation approaches and thus facilitate their systematic comparison. They are based on our experience with UCP and have been discussed and refined with fellow researchers. A dedicated Website where key-feature-based classifications of 14 UI generation approaches, provided by the respective developing scientists, is currently under construction[10].

[10] http://ui-gen.ict.tuwien.ac.at

2 Background and State-of-the-Art

This chapter starts with providing background information on model-driven software engineering and model-based UI development (MBUID) in Section 2.1. In particular, we will introduce the Model Driven Architecture (MDA) as a conceptual approach for model-driven software development, and the Cameleon Reference Framework (CRF), as a conceptual approach for the classification of UI generation approaches. Both conceptual approaches define several levels of abstraction for the models involved with the aim to support platform-independent development (e.g., multi-device UI development). We present the levels of abstraction defined in MDA and the CRF, respectively, and discuss their relationship based on their different notions of platform.

In Section 2.2 we introduce state-of-the-art UI generation approaches with the focus on design-time UI generation. We classify these approaches based on a set of properties that we defined as "key features". In particular, we present how these approaches relate to MDA and the CRF and how they support multi-device UI generation.

Our state-of-the-art comparison allows us to identify the Unified Communication Platform (UCP) approach as the most suitable one for implementing the Interactive Multi-device GUI Generation approach proposed in this doctoral dissertation. We provide background information on UCP in Section 2.3 to facilitate the understanding of how we implemented our conceptual approach and to clarify what the contributions of this doctoral dissertation are.

2.1 Conceptual Background on Model-driven UI Generation for Multiple Devices

This section introduces two conceptual approaches for multi-device UI generation – the Model Driven Architecture and the Cameleon Reference Framework. We particularly focus on their respective definition of platform and relate them for the purpose of multi-device UI generation. For a better understanding of what the concepts for model-driven UI generation are based upon, let us start with taking a look at their historical background – model-based system development.

Model-based system development assumes the use of models during development. In this context, models are primarily used to facilitate the communication between experts in different domains. They are not directly involved in the development process and are thus an optional add-on [BCW12]. Model-based development has been used for UI development since the 1980s [Sze96, MPV11], aiming for high-quality UIs with reduced development effort. In the 1990s,

model-based UI development approaches already specified architectures, development methods based on models, and how these models have to be refined to achieve running applications, in addition to exploiting their communicative value (e.g., Mobi-D [Pue97] or Trident [BHLV95]). Computer-aided software engineering (CASE) approaches added tool support for model-based software development.

CASE approaches were the first to support *model-driven* software engineering, which combines models and automated transformations [Sch06, BCW12]. Models are used to abstract from specific implementations and to facilitate the interoperability between systems. Automated transformations are applied to derive specific implementations in a (semi-)automated way. In 2003, the Object Management Group (OMG) proposed the Model Driven Architecture (MDA) as a standard for model-driven engineering [MM03].[1] MDA is a special kind of model-driven engineering, which uses the concept of a platform to encapsulate implementation-specific information. This information is only considered during the transformation from the so-called Platform Independent Model to the so-called Platform Specific Model.

MDA has been designed as a general-purpose architecture for model-driven software development and is, therefore, also suitable for model-driven UI development [OP07]. MDA even provides a conceptual architecture for multi-device UI development, as its notion of a platform is open and allows modeling a device with software and hardware characteristics as platform. The platform definition of MDA focuses on functionality and does not explicitly enforce the inclusion of software or hardware characteristics. Ignoring hardware characteristics, like screen size and resolution, however, decreases the usability of the resulting UIs [AVCF+10].

The Cameleon Reference Framework (CRF) [CCT+03] translates the conceptual approach of MDA to the model-based UI development domain. It provides a platform notion that includes software and hardware characteristics explicitly and allows for the classification of UI generation approaches. We introduce and relate MDA and CRF in the following, focusing on their different notions of platform and their implications for multi-device UI generation, based on [RMB13].

2.1.1 Model Driven Architecture

The OMG's Model Driven Architecture (MDA) specifies a generic approach for model-driven software engineering [MM03]. MDA distinguishes four different levels of abstraction. These are (from abstract to concrete):

1. the Computation Independent Model (CIM),
2. the Platform Independent Model (PIM),
3. the Platform Specific Model (PSM) and
4. the Implementation (or Implementation Specific Model – ISM)

The *Computation Independent Model* is on the highest level of abstraction and, therefore, both computation and platform independent. It models the context in which the system to be built will be used, specifying exactly what the system is expected to do. Such models can be used to capture requirements on the system and are sometimes referred to as domain or business models in the context of MDA.

[1] OMG Object and Reference Model Subcommittee – `http://ormsc.omg.org/mda_guide_working_page.htm`

The *Platform Independent Model* specifies the system to be built from a platform independent viewpoint, which "focuses on the operation of a system while hiding the details necessary for a particular platform. A platform independent view shows that part of the complete specification that does not change from one platform to another" [MM03]. So, a PIM can be mapped onto different platforms.

The *Platform Specific Model* specifies how a system uses a particular platform. This model needs to specify all details necessary to derive the implementation.

The *Implementation Specific Model* (ISM) can be operationalized and represents the implementation of the system (e.g., in the form of source code).

MDA applies *Model Transformations* to transform PIMs to PSMs. Such transformations need to provide the additional information required to produce the PSM or gather it from the Platform Model during the transformation process. MDA does not propose automated transformations between CIM and PIMs and does neither propose transformations between PSM and implementation.

Platform is a fundamental concept in MDA, as the promises of technology obsolescence, rapid portability, increased productivity, shorter time-to-market, consistency and reliability of produced artifacts [Tru06] are based on the abstraction from a certain platform. The same concept of abstracting from details is required for multi-device UI generation. In particular, multi-device UI generation requires the abstraction from both software and hardware characteristics to achieve a good level of usability [AVCF+10]. This is potentially supported by the MDA's platform definition:

> *"A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented [MM03]."*

A platform is represented as a *platform model*, which is defined as:

> *"A platform model provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in a platform specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application. A platform model also specifies requirements on the connection and use of the parts of the platform, and the connections of an application to the platform. A generic platform model can amount to a specification of a particular architectural style [MM03]."*

The MDA platform definition focuses on functionality and potentially supports the consideration of software and/or hardware characteristics, but it does not enforce either. This means that approaches that encapsulate software characteristics only in their platform model (e.g., UI toolkits) are MDA-compliant. Such approaches are not able to consider device properties during the transformation process, which typically results in an unsatisfying level of usability of the generated GUIs. The reason is that a functional platform definition focuses on providing the functionality to put a specific implementation into operation, but it does not consider properties that are important for the interaction (e.g., available screen space or mouse- or finger-based interaction) and thus to achieve a satisfying level of usability.

2.1.2 Cameleon Reference Framework

Model-driven UI Development uses models to specify all aspects involved. Such approaches typically refine high-level interaction models over different levels of abstraction to source code that implements the UI. The Cameleon Reference Framework (CRF) allows for the classification of UIs and generation approaches that support multiple targets, or multiple contexts of use in the field of context-aware computing [CCT+03], with the intention to ease understanding and comparing them.

In particular, the CRF introduces four levels of abstraction. These are (from abstract to concrete):

1. Tasks & Concepts Level,
2. Abstract User Interface Level,
3. Concrete User Interface Level and
4. Final User Interface Level.

The *Tasks & Concepts* level contains interaction models, e.g., of tasks to be performed by the user and the system to be built, and models of the domain of activity.

The *Abstract User Interface* level typically contains a presentation and a dialog model, which render the domain concepts into canonical expressions that are independent from any concrete interactors available on a certain platform (i.e., modality- and device-independent).

The *Concrete User Interface* level contains models with concrete interactors that specify the look and feel of the user interface, but are still independent from a certain toolkit (i.e., modality- and device-dependent).

The *Final User Interface* is the UI source code that can be compiled and run.

MBUI typically applies model-to-model transformations between the upper three levels of abstraction and model-to-code transformations between Concrete and Final UI level.

The notion *platform* here relates to UI development and is defined as part of the *context of use* (together with the *user* and the *environment*). In particular, the platform consists of:

> *"a set of hardware (e.g., processor, screen, and mouse) and software resources (e.g., operating system, technological space) that function together to form a working computational unit whose state can be observed and/or modified by a human user. Single resources (processor, peripheral devices etc.) are unable, individually, to provide this functionality.*
>
> *A platform may be either elementary or form a cluster of platforms. Synonyms: target platform."*[2]

This notion explicitly includes software and hardware resources and interaction properties as part of a platform in contrast to the MDA platform definition. The next subsection establishes a correspondence between the MDA and CRF levels, as intended by Calvary et al. [CCT+03] and already applied for the User Interface Extendable Markup Language (UsiXML) in [Van05].

[2] see `http://www.w3.org/wiki/Model-Based_User_Interfaces`

2.1.3 Correspondence between MDA and CRF

MDA was defined in the context of software engineering. Examples provided for platform are operating systems, programming languages, databases, middle-ware solutions or user interfaces [Tru06]. In the case of user interfaces, the platform can be interpreted as the toolkit that implements the UI, but since the MDA platform definition is very generic, it can also be interpreted as software, hardware and interaction properties (e.g., a smartphone with a certain operating system, physical characteristics like memory or screen size and touch-based interaction).

It is recommendable to refine the MDA platform definition for a given application domain to facilitate its applicability and avoid misunderstandings. The CRF platform definition can be seen as such a refined definition for the MBUID domain. A CRF platform consists of hardware (physical) properties (e.g., screen size and resolution, supported interaction modalities) and software properties (e.g., toolkits like Java Swing or HTML that implement a certain modality).

Such a platform definition is suitable to encapsulate all characteristics of a device that are relevant in the context of GUI generation. This potentially supports multi-device UI development and allows for establishing a clear correspondence between MDA models and CRF levels, as intended by Calvary et al. [CCT+03] and illustrated in Table 2.1. This correspondence scheme focuses on UI models only and has already been applied during the development of UsiXML [Van05]. All other models that can be specified in the CRF (e.g., user or environment) can be classified as Computation Independent Models in the context of MDA.

Table 2.1: Correspondence Scheme

MDA		CRF
CIM	↔	Tasks & Concepts Level
PIM	↔	AUI Level
PSM	↔	CUI Level
ISM	↔	FUI Level

This correspondence requires the use of the CRF platform definition, which we refer to as platform from hereon.

Computation Independent Models (CIMs) model the interaction between the user and a system, without any consideration of a specific platform. High-level interaction models (e.g., ConcurTaskTree Models [PMM97] or Discourse-based Communication Models [FKH+06]) can be used to specify the flow of interaction between the user and the system platform-independently. Together with models that specify the concepts that can be manipulated by a user in a specific application domain (i.e., domain models). Both are on the Tasks & Concepts Level of the CRF.

Platform Independent Models (PIMs) correspond to the AUI Level of the CRF, as AUI models are by definition independent of any concrete interactors of a certain platform.

Platform Specific Models (PSMs) correspond to the CUI Level, as both models already contain platform-specific information.

Finally, the PSM is transformed to the source code that implements the UI. Thus, the implementation (or Implementation Specific Model (ISM)) corresponds to the FUI Level.

So, what does this correspondence mean for models on Tasks & Concepts Level, and implicitly also for AUI models derived from them? Models on the Tasks & Concepts and the AUI Levels

need to be platform-independent, i.e., independent of any platform model properties. In other words, such models must not restrict the rendering possibilities that are potentially supported through the Platform Model.

In particular, it means for the high-level interaction model that *all information that can be exchanged at a certain point in time needs to be modeled as concurrently available*, to allow for the generation of several GUI models from the same interaction model. This means that such models assume a "potentially infinite" screen. Specifying information as concurrently available allows for splitting it according to platform constraints during the transformation process (e.g., tailoring the GUI to a limited screen). Doing it the other way round (i.e., combining information) is hard to achieve as it requires the analysis of dependencies between the exchanged information.

For creating the domain model it means that such a model needs to define all concepts of the application domain that can potentially be communicated about with the user, regardless of whether they are used by a certain interaction model (that may be device-dependent) or not. The reason is that different platforms may only support a subset of the specified interaction. For example, a system may provide elaborate configuration options on its desktop GUI, which are not available on the corresponding smartphone GUI.

2.2 State-of-the-Art UI Generation Approaches

This section presents an overview of state-of-the-art UI generation approaches. We start with discussing the difference between run-time and design-time UI generation to motivate our choice for the latter in this doctoral dissertation. Subsequently, we present and compare design-time UI generation approaches that support multi-device UI generation based on a set of UI generation approach properties that we defined as so called *key features*.[3] This comparison allows us to motivate our choice for the Unified Communication Platform (UCP) as a basis for the implementation of the conceptual approach of this doctoral dissertation.

2.2.1 Design-Time UI Generation

Design-time UI generation transforms all models to executable source code at design-time and does not use the models at run-time. Run-time UI generation, in contrast, does not generate the source code of the system in advance, but interprets the models at run-time.

Run-time UI generation enables the designer to directly see the impact of modifications on the input models. This is especially helpful if different alternatives should be explored (e.g., the combination of different modalities in multi-modal UI development), as it allows the discussion of changes while giving the targeted audience the opportunity to experience the UI.

Design-time UI generation, on the other hand, gives the designer control over the resulting UI through typically providing models of the same UI on different levels of abstraction. This allows the designer to customize the resulting UI on different levels of abstraction and offers better predictability of the resulting UI (e.g., an impression of the final GUIs look & feel) than run-time generation, typically at the price of less flexibility regarding the exploration of alternatives.

[3] The basic ideas of our key features is to facilitate the systematic comparison of UI generation approaches, the complete key feature specification is attached to this doctoral dissertation as Appendix A.

Table 2.2 classifies 15 state-of-the-art UI generation approaches according to their generation time, using a double horizontal line to separate approaches that apply generation at run-time, design-time or run- and design-time generation. We selected the generation time as main classification property because all model-driven UI generation approaches have to apply transformations at a certain point in time. All approaches in our comparison are still under development/maintenance and available for researchers. This is important because building on an existing approach saves researchers the time required for re-implementation of previously developed and tested concepts and provides the basis for more advanced research.

Table 2.2: UI Generation Approach Overview

		Run-time	Design-time	Modalities	Highest CRF level for automatic UI Generation
1	**Makumba**	x		GUI only	CUI
2	**MASP**	x		MM	CUI
3	**ProFi**	x		MM	AUI
4	**SmartMote**	x		MM	CUI
5	**IFML**		x	GUI only	AUI
6	**MANTRA**		x	GUI only	AUI
7	**MML**		x	GUI only	CUI
8	**MBUE**		x	GUI only	Tasks & Concepts
9	**TERESA**		x	GUI only	Tasks & Concepts
10	**UCP**		x	GUI only	Tasks & Concepts
11	**UsiXML**		x	MM	Tasks & Concepts
12	**AALuis**	x	x	MM	Tasks & Concepts
13	**MARIAE**	x	x	MM	Tasks & Concepts
14	**OO-Method**	x	x	GUI only	CUI
15	**UsiComp**	x	x	GUI only	Tasks & Concepts

MM – Multi-Modal

The overview shown in Table 2.2 illustrates that the generation time typically correlates with which types of UIs are supported (i.e., GUIs or multi-modal UIs), and the input models' level of abstraction. Run-time UI generation is typically applied to generate multi-modal UIs, starting from lower levels of abstraction than design-time generation, which is typically applied for GUI generation. Better support for exploring alternatives makes run-time UI generation suitable for finding an appropriate combination of modalities for a certain application, whereas creating highly customized UIs for different devices using run-time generation takes a lot of effort, because such approaches typically use input models on CUI level. Input models on CUI level are not suitable for multi-device UI generation, as they are already device-specific.

Interestingly, there are also approaches that support generation both at run- and design-time. Table 2.2 shows that their level of abstraction varies between the highest and the lowest CRF level, depending on their focus. Support for distributed UIs, as provided in MARIA [MP11], for example, is hard to achieve with pure design-time UI generation, because it would require the calculation of all possible situations at design-time. Metawidget and OO-Method start from a low level of abstraction as they support fully-automated UI generation on an industrial scale. Automated generation from a higher level of abstraction (e.g., interaction models on tasks &

concept level) typically involves the completion of lower level details (e.g., style and layout) based on heuristics. This makes it hard for the designer to control understand the connection between specification and final result [MHP00].

Run-time generation approaches like Makumba[4] [KB13], MASP[5] [RLS+11], MINT[6], ProFi [HSW12] or SmartMote [SBD+10] do not support multi-device UI generation in an automated way, as they require the manual creation of a device-specific GUI model. The remainder of this section concentrates on design-time UI generation approaches and how they achieve multi-device UI generation, because the conceptual approach of this doctoral dissertation aims for multi-device GUI generation at design-time. In particular, Interactive Multi-device GUI Generation automatically transforms a device-independent high-level interaction model into a device-specific Screen Model, which can be customized by the designer before the GUI source code is generated.

In the following, we present the 11 approaches listed in Table 2.2 that apply UI generation at design time in more detail. We will focus on their properties *models*, *transformation method* and *multi-device UI generation*. The used properties are a sub-set of UI generation approach properties that we defined as *key features*.[7] Our key features support the classification of UI generation approaches and facilitate their systematic comparison. All 11 key-feature-based classifications of UI generation approaches used for our comparison have been provided by the researchers that develop the corresponding approach. The basic idea was to directly ask the developing researchers for information to avoid misunderstandings. So far, we collected 17 key feature descriptions in total, that will be available on our Web-site on UI generation[8] in the future.

2.2.1.1 IFML

The Interaction Flow Modeling Language[9] (IFML) supports the creation of visual models of user interactions and front-end behavior in software systems, independently of a certain execution platform and has been adopted as a standard by the OMG[10] in March 2013. IFML uses Business Process Model and Notation[11] (BPMN) models on the CIM Level, which are however, not directly used for GUI generation. IFML itself is a PIM/AUI-level language. The IFML standardization document includes a set of guidelines for the mapping to PSMs, such as Java, .Net WPF, and HTML.

The Web Extension of IFML, called WebML[12] [CBF09], provides a PSM/CUI for the Web, which extends and seamlessly maps to the general purpose IFML concepts. WebML also includes a presentation model as ISM/FUI Model. The transformation process, starting from IFML models, is supported through the WebRatio tool-suite [ABB+08]. IFML, in particular, provides an open source, Eclipse-based IFML Editor, which supports the integration of the generated GUI with the back-end specification in UML, with fUML and Alf.

[4] http://www.makumba.org
[5] http://masp.dai-labor.de
[6] http://www.multi-access.de
[7] Please refer to Appendix A for the full list.
[8] http://ui-gen.ict.tuwien.ac.at
[9] http://www.ifml.org/
[10] http://www.omg.org/spec/IFML/
[11] http://www.bpmn.org/
[12] http://www.webml.org

According to [IFM13], IFML supports multi-device GUI generation through the definition of rules at the PIM level for self-adaptation of user interfaces depending on the device, screen size, or location.

2.2.1.2 MANTRA

MANTRA supports the semi-automated generation of multiple consistent GUIs for the same application [Bot11]. In particular, it supports the generation of Web-service front-ends based on a device-independent abstract UI model. Such abstract UI models are gradually refined to source code in three transformation steps. The first step transforms the device-independent abstract UI model into a device-specific one. A first clustering of the abstract interaction elements is done based on heuristics like the number of UI elements in each presentation or the nesting level of grouping elements, but the authors state that this clustering is typically refined manually according to the screen size of a given device. The subsequent steps transform this already device-dependent AUI model into a CUI model and subsequently into source code.

This approach supports automated UI generation for different software platforms (i.e, a Web and a Dotnet front-end), but hardware device-characteristics have to be considered manually through tailoring the AUI for a certain device.

2.2.1.3 MML

The Multimedia Modeling Language (MML) [PH11] addresses model-driven development of interactive multimedia applications, i.e., applications where the UI includes individual media elements such as interactive animations, graphics, images, audio, video and 3D graphics. Examples of such elements are interactive maps or presentations in an e-learning tool, interactive 3D models of products in e-commerce applications, or characters in computer games. MML uses concepts from model-driven UI development and extends them with support for complex media elements to integrate and foster systematic collaboration between software designers (for the application logic), user interface designers (for the overall interaction and usability), and media designers (design of specific media types). MML supports the integration of professional media authoring tools by generating code skeletons that can be directly loaded, processed and completed with such tools. In this way, the advantages of model-driven engineering (systematic development, platform-independence) and authoring tools (powerful support for visual design) are combined. MML comes with model transformations to generate code for different target platforms. It supports generation of Flash/ActionScript application skeletons to be finalized in the authoring tool Adobe Flash. Other target platforms include Java, SVG/JavaScript, and Flash Lite for mobile devices.

MML supports multi-platform GUI generation in the sense of toolkit independence. It does not provide support for multi-device GUI generation, as the input model is already on CUI level, but rather focuses on support for advanced multi-media interaction.

2.2.1.4 MBUE

"Useware is a generic term introduced in 1998 that denotes all hard- and software components of a technical system which serve its interactive use. The main idea of the term Useware is the

focus of technical design according to human abilities and needs"[13]. The model-based UseWare Engineering (MBUE) approach supports *Useware* compliant UI generation for applications in process-oriented industrial automation environments [MBS11]. Its model-based tool chain is structured according to the different abstraction levels of the CRF.

The initial analysis phase is mapped to the Tasks & Concepts level and uses the Userware Data Description Language (useDDL) together with the Useware Markup Language (useML) for task models. The AUI level is mapped onto a part of the design phase and uses the Dialogue and Interface Specification Language (DISL) to specify modality-independent UIs. The CUI level is mapped onto another part of the design phase and uses the User Interface Markup Language (UIML) that provides a generic (graphical) vocabulary to specify the UI. The transformations are supported by tools and can be performed (semi-)automatically, resulting in a FUI in either JAVA, HTML or C++.

MBUE relies on the CRF platform definition and applies task annotations in its useML task model that assign tasks to a certain context of use (i.e., platform and/or user and/or environment). In this way, MBUE supports the automated tailoring of the UIs to a certain context of use, based on one platform-independent (annotated) task (i.e., useML) model.

2.2.1.5 TERESA

TERESA (Transformation Environment for inteRactivE Systems representAtions) uses a so-called "One Model, Many Interfaces" approach to support model-based multi-modal UI development for multiple devices from the same ConcurTaskTree (CTT) model [MPS04]. This approach defines a platform as *"a class of systems that share the same characteristics in terms of interaction resources (e.g., the graphical desktop, PDAs, mobile phones, vocal systems). Their range varies from small devices such as interactive watches to very large flat displays"* [MPS04].

However, the corresponding transformation method requires the manual refinement of the platform-independent CTT model to different platform-dependent System Task Models (e.g., a desktop, a cellphone or a voice System Task Model, still specified in CTT). This already platform-dependent CTT model is subsequently refined to an AUI, a CUI and finally the FUI for the corresponding platform.

2.2.1.6 UCP

The Unified Communication Platform[14] (UCP) uses Discourse-based Communication Models [FKH+06, Pop12] to model the communicative interaction between two parties (e.g., user and system) modality and device independently. Its UI generation approach UCP:UI supports the automated generation of device-tailored Window / Icon / Menu / Pointing Device (WIMP) UIs based on such models.

Communication Models are CIMs and can be transformed automatically to a UI Behavior Model that resides on the AUI level [PFA+09] and is still platform-independent, and to a Screen Model that is tailored to a certain device. The Screen Model is platform-specific (i.e., corresponds to the CUI level of the CRF) and transformed to source code that represents the FUI/ISM in the last transformation step.

[13] http://en.wikipedia.org/wiki/Useware

[14] http://ucp.ict.tuwien.ac.at

UCP uses a so-called *Application-tailored Device Specification* [KRF+09] to model software and hardware characteristics of a certain target device (e.g., smartphone or tablet PC) together with properties regarding the interaction for a specific application. For example, a certain application may require the use of a tablet PC with a pen as pointing device, which may require a different set of interaction widgets compared to if the device was operated with a finger. The GUI is automatically optimized according to the constraints defined in the Application-tailored Device Specification when the Communication Model is transformed to the Screen Model [RPK+11b].

2.2.1.7 UsiXML

The USer Interface eXtensible Markup Language[15] (UsiXML) is an XML-compliant markup language that supports the descriptions of UIs for multiple contexts of use (e.g., GUIs, multi-modal UIs, distributed UIs, etc.). UsiXML specifies different meta-models, including UI models on all CRF levels, to support multi-platform UI development, and is MDA compliant by definition [Van05]. UsiXML proposes a conceptual methodology that allows for different development paths [LV09] and provides the basis for different UI generation approaches and tools [Van08].

UsiXML is currently available in version 2.0, which provides graphical editors for the Task, the Domain and the Context Models. UI customization is supported through a graphical editor for the AUI and the CUI. UsiXML is not limited to design-time UI generation, as it does not provide an implementation of a transformation method. UsiComp [GFCDC+12], for example, uses UsiXML models for run-time UI generation.

2.2.1.8 AALuis

The Ambient Assisted Living User Interface (AALuis) approach supports semi-automatic, multi-modal UI generation for different target devices [MMG+13]. It uses a task model in CTT notation to specify the interaction between the user and the system and/or service. Services are integrated as internal Open Services Gateway initiative (OSGi) bundles or as external Web services defined in Web Service Description Language (WSDL). A separate XML file defines the binding between CTT tasks and service functions and allows a clear separation of concerns. The task model, which is developed manually for a specific service, provides the starting point for the automatic UI transformation process.

Active tasks are transformed into a modality and device independent AUI, the modality and available device dependent CUI and finally the Renderable User Interface (RUI), which can be rendered by a specific I/O device. The applied user model and the current context model support the device selection. AALuis supports multiple modalities and enables, amongst others, the on-the-fly avatar video generation for the concrete rendered UI at run-time. According to [MMG+13], multi-device GUI generation is supported through device-specific layout templates and transformation rules.

[15] `http://www.usixml.org`

2.2.1.9 MARIAE

The MARIA Environment[16] uses task models (represented in the ConcurTaskTrees notation) and UI models (specified in the MARIA language[17]) for the design and development of interactive applications, based on Web services, for various types of platforms (desktop, smartphones, vocal, multimodal, etc.) [PSS09b].

This approach supports two development paths with different entry points. First, it is possible to associate an existing Web service with a task model of the interactive application, and then to generate multiple UIs for various platforms [PSS10b]. Similarly to MBUE, MARIAE requires Web-service annotations in both cases, to achieve multi-device UI generation [PSS10a]. Second, it is possible to adapt an existing GUI for a new device through the semantic redesign method. According to [PZ10], it is possible to automatically generate a Concrete UI Model for an existing HTML Web-page, adapt it for the new target device and generate the corresponding HTML code. These adaptations are conceptually similar to the adaptation rules used by IFML.

MARIAE supports design-time GUI customization on AUI and CUI levels through a graphical tree editor. In addition, MARIAE supports run-time adaptations in terms of multi-modality, generating an appropriate FUI on demand for a given context of use. This enables MARIAE to support UI plasticity, migratory and distributed UIs.

2.2.1.10 OO-Method

OO-Method is a conceptual schema-centric software development approach [PGIP01, PM07]. It follows the paradigm of Conceptual Modeling [ELP11] by creating a conceptual schema consisting of four models: structure, functionality, behavior and presentation. The structural, functional and behavior models are UML-compliant and the presentation model is expressed in UsiXML. These four models together represent the "conceptual program", that a conceptual program compiler transforms into a running software application, either at design- or run-time. Tool support for OO-Method is provided by IntegraNova, an industrial tool from CARE Technologies[18]. This approach is especially well suited to generate database applications like organizational systems.

OO-Method relies on a presentation model at CUI level, which needs to be adapted manually according to the hardware constraints of a certain platform.

2.2.1.11 UsiComp

The UsiComp framework supports the reification of task and domain models to GUI source code under the consideration of a context and a quality model, according to [GFCDC+12]. It applies automated transformations to derive the GUI for a certain device at design-time, but additionally supports run-time GUI generation to enable GUI adaptations according to context changes. In particular, UsiComp is able to generate the GUI for a new device as soon as the device becomes available, if this new device corresponds to an already available platform definition.

[16] `http://giove.isti.cnr.it/tools/MARIAE/`
[17] `http://giove.isti.cnr.it/tools/MARIA/`
[18] `http://www.care-t.com`

UsiComp supports multi-device UI generation through specific transformations. It supports manual GUI customization on CUI level through supporting the additional input of a GUI mockup done in Balsamiq[19].

2.2.2 Discussion

This subsection discusses the current state-of-the-art presented above with respect to the research problems of this doctoral dissertation.

Support for fully automated generation of different device-specific GUIs (Research Problem 1) requires a device-independent model as a starting point. Such models are typically either high-level interaction models (e.g., Task or Communication Models) or device-independent UI models (e.g., AUI models). These models have to be independent of the corresponding platform, which has to be specified explicitly to allow for automated multi-device GUI generation.

MANTRA and TERESA, for example, provide such models, but use a platform model that considers software (i.e., toolkit) characteristics only. This means that their input models require manual adaptation according to the screen size of a given device. Both approaches use a specific meta-model on different levels of abstraction. MANTRA manually tailors a platform-independent AUI (PIM) to a platform-specific AUI (PSM) and TERESA tailors a "nomadic task model" (a PIM) manually to a "system task model" (a PSM), both specified in CTT. These approaches additionally show that instances of task or AUI models do not have to be on the corresponding CRF level by definition, but that this assignment depends on the characteristics, and in particular on the platform model, of a given UI generation approach.

Current state-of-the-art UI generation approaches typically support automated multi-device UI generation through the use of annotations (e.g., MARIAE, MBUE) or platform-specific adaptation rules (e.g., IFML), where such annotations or transformation rules have to be specified manually for each platform. Alternatively, the platform-tailoring can be automated according to given optimization objectives, as applied by UCP [RPK+11b]. Both options provide a feasible solution for research problem number one of this dissertation (i.e., automated multi-device UI generation), though automated tailoring saves the time and effort required for creating annotations manually, and allows for the automatic exploration of design alternatives.

Support for iterative manual GUI adaptations through the designer is typically not available in current state-of-the-art UI generation approaches, because keeping the models consistent is left to the designer, so that the customizations are typically lost in case of re-generation [MPV11]. Pleuss et al. recently showed how customizations can be re-applied successfully in case of re-generation [PWB13]. This doctoral dissertation proposes a similar storage mechanism together with customization persistence through custom transformation rules, embedded in a tool-supported process for iterative and incremental GUI development to tackle this issue.

Based on these research questions and our state-of-the-art research, an earlier version of the Unified Communication Platform[20] was selected as a basis for the prototypical implementation of the conceptual approach of this doctoral dissertation and its evaluation for two major reasons. First, UCP provides Discourse-based Communication Models, which allow representation of communicative interaction between two parties, together with the concepts that are

[19] http://www.balsamiq.com/

[20] http://ucp.ict.tuwien.ac.at

exchanged between them. Support for these aspects has been identified as important to enable UI generation [AVPP11]. Second, UCP already provided an earlier UI generation approach [Kav11, Pop12, KFK09b], which relied on a proprietary and accessible transformation engine and used the concept of an application-specific device model. This facilitated the implementation of our conceptual approach that treats device-tailoring as an optimization problem. SUPPLE supported automated GUI optimization for motor-impaired users at run-time [GW04], in contrast to our approach, which employs optimization in the context of automated GUI-tailoring for a given device at design-time. Our new approach facilitates the exploration of design-alternatives in contrast to the use of annotated interaction models. Integrating our new concepts directly in UCP, furthermore, allowed us to provide the designer with customization possibilities that go beyond providing additional input [AVVP09] (e.g., persistent layout customization on CUI level).

2.3 The Unified Communication Platform

This section presents an overview of the earlier version of the Unified Communication Platform (UCP), which provided the basis for the prototypical implementation of the conceptual approach of this doctoral dissertation and its evaluation. This earlier version supported the semi-automatic transformation of Discourse-based Communication Models to Java Swing GUIs. It did not provide a Screen Model and thus no automated device-tailoring, which made the exploration of design alternatives a labor-intensive and error-prone task. In addition, no tool-supported processes for multi-device GUI development and customization were available then.

Figure 2.1 illustrates the three major modules of UCP. These modules have already been created for the earlier version and extended for the current one. The first module is the Discourse-based Communication Model (UCP:CM), which supports the specification of communicative interaction between two parties [FKH+06, Pop12]. The second module is a UI generation framework (UCP:UI), which semi-automatically renders GUIs for different target devices. The third module is a run-time environment (UCP:RT), which integrates the generated UIs (UCP:UserServices) and the application back-end (UCP:MachineServices) [Pop12].

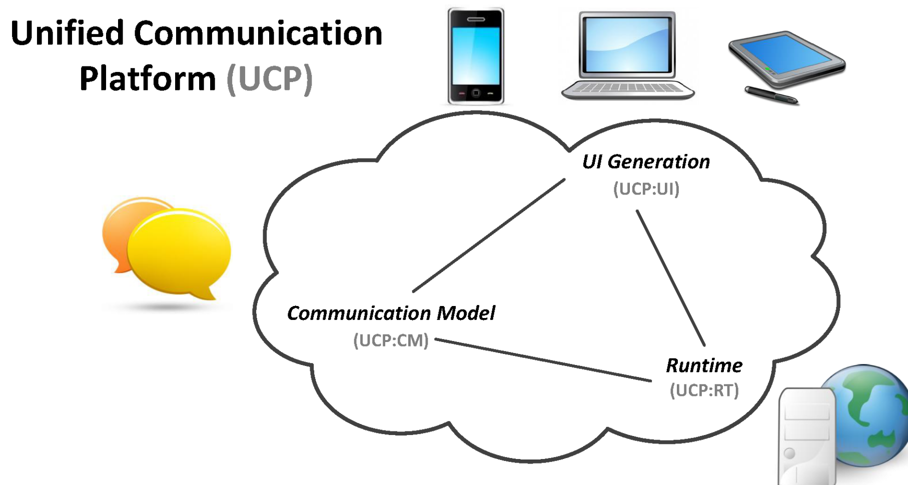


Figure 2.1: The Unified Communication Platform Overview

2.3.1 Discourse-based Communication Models (UCP:CM)

Discourse-based Communication Models have been developed at the Institute of Computer Technology of the Vienna University of Technology, to provide a unified way of specifying human-machine and machine-machine interaction [FKH+06, Pop12]. Such models consist of a Domain-of-Discourse Model, an Action-Notification Model and a Discourse-Model, which integrates these three models to a Discourse-based Communication Model.

This subsection provides background information on these models as they are used as a basis for the concepts and their implementation presented in this doctoral dissertation. Additionally, we introduce two new Discourse Relations that have been added to enrich the semantic expressiveness of Discourse Models for UI generation.

We use a simple bike rental scenario to illustrate our modeling approach and the previous GUI generation approach [KBFK08, Kav11] of the earlier UCP version. This simple scenario requires unregistered users to register in order to login, to rent and subsequently to return a bike.

2.3.1.1 Domain-of-Discourse Model

The Domain-of-Discourse (DoD) model defines the concepts that the two interacting parties can "talk" about. These concepts do not necessarily have to be used by the application logic, in case that one interacting party is a machine, but rather define a common meaning for the information that is exchanged. These concepts can also be mapped to a database schema before/after being exchanged to be persisted. The DoD model is typically represented as a UML or Ecore class diagram.

Figure 2.2 shows the DoD model (defined as Ecore model) for our bike rental scenario. This model defines the concept `User` with the attributes `userName` and `password`, for example, together with further concepts that are required to exchange information between the user and the application.

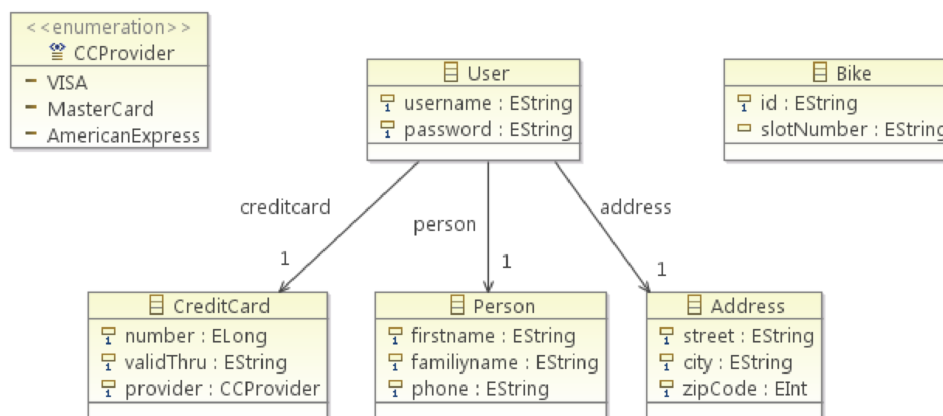


Figure 2.2: Bike Rental Domain-of-Discourse Model

2.3.1.2 Action-Notification Model

The Action Notification Model (ANM) represents Actions and Notifications that can be performed by the two interacting parties [Pop12]. Examples of predefined *Actions* are `get` and `set`, which can

be used to specify getting and setting a specific variable. An example of a predefined *Notification* is `presenting`, which can be used to present the value of a certain variable. UCP provides a set of predefined Actions and Notifications, the so-called *basic* ANM (for details see [Pop12]), as default. This *basic* ANM can be extended to provide the functionality required for a specific application.

Figure 2.3 shows the additional Actions and Notifications defined for the bike rental application. Examples of such specific Actions are the `startRegistration` Action used to trigger the registration process and the `confirm` Action with the parameter `selectedBike` used to confirm the bike selection before the bike is actually assigned as rented to a certain user.

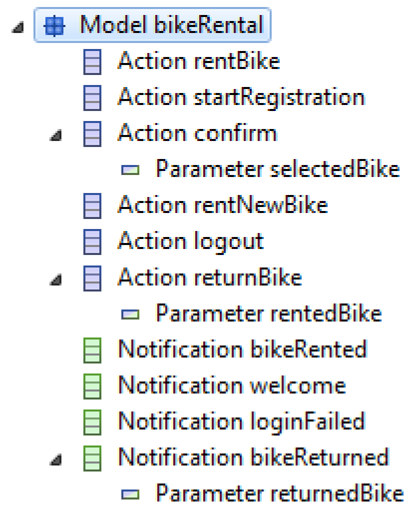


Figure 2.3: Bike Rental Action Notification Model

Examples of specific Notifications are `loginFailed`, which informs about an unsuccessful login attempt, and the `bikeReturend` Notification, which informs that the bike specified through the `returnedBike` parameter has been returned successfully.

2.3.1.3 Discourse Models

Discourse Models specify high-level communicative interaction between two agents primarily based on discourses in the sense of dialogues. In case of GUI generation, one of these agents is the `User` and the other one the `System` (see Figure 2.4).

Figure 2.4: Interacting Agents

A small excerpt of the Discourse Model for our bike rental scenario is shown in Figure 2.5. This excerpt models the interaction between the User and the System during the collection of registration data (i.e., username and password, personal details and credit card information).

The basic building-blocks of Discourse Models are *Communicative Acts*, a generalization of Searl's Speech Acts [Sea69]. Communicative Acts are depicted as rounded rectangles in the graphic

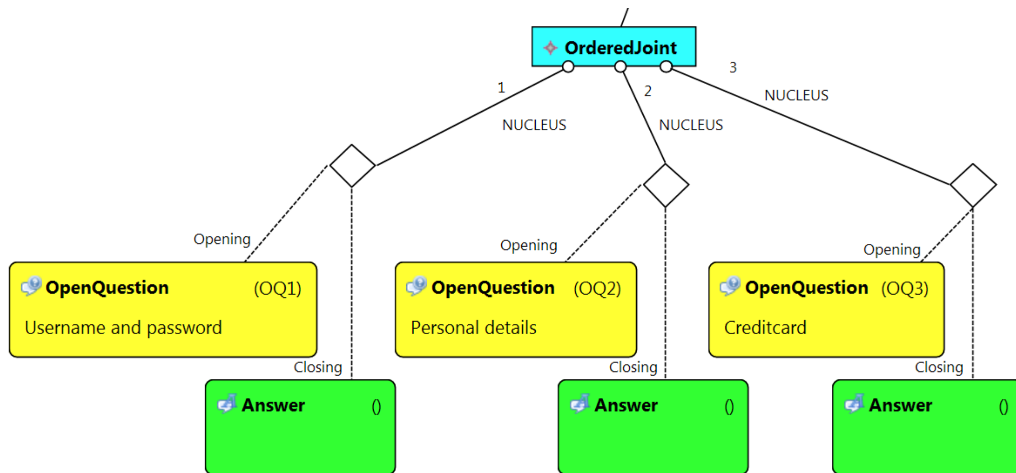


Figure 2.5: Bike Rental Excerpt with Ordered Joint

notation of Discourse Models (see Figure 2.5) and each Communicative Act is assigned to an agent, represented through its fill color (green/dark for User, and yellow/light for System).

*A*djacency Pairs relate one opening and zero to two closing Communicative Acts and model typical turn-takings in a conversation (e.g., Question–Answer or Offer–Accept/Reject). Such Adjacency Pairs have been derived from Conversation Analysis [LFG90] and are represented through diamonds in Figure 2.5.

Additionally, *Discourse Relations* can be used to link Adjacency Pairs or other Discourse Relations and to model more complex discourse structures. We distinguish two kinds of such relations: Relations derived from Rhetorical Structure Theory [MT88] – *RST Relations* – and relations that primarily control the flow of events – *Procedural Constructs*. An example of an RST Relation is the `Background` relation, which relates the core information of a Nucleus branch and additional information provided in a Satellite branch. An example of a Procedural Construct is `Sequence`, which relates two or more Nucleus branches to be executed in an explicitly specified sequential order. For more details please see the doctoral dissertation of Roman Popp [Pop12].

The connection between the Discourse, the Domain-of-Discourse and the Action-Notification Models is established through the *propositional content* specified for a Communicative Act [PR11, Pop12]. The propositional content references concepts defined in the ANM and the DoD models and specifies which Actions/Notifications have to be performed by the interacting party that the corresponding Communicative Act is assigned to, and which DoD concepts are involved.

The propositional content is specified in an SQL-like grammar. The propositional content of the leftmost OpenQuestion (OQ1) in Figure 2.5 is defined as `basic::get one bikeRental::User::newUser for basic::set bikeRental::User::newUser`. This specification refers to the `set` and `get` Action defined in the default ANM and to the `User` concept specified in the `bikeRental` DoD model (see Figure 2.2) and specifies that the data collected through the OpenQuestion is to be stored in a variable of the type `User`, with the name `newUser`, by the System (see the yellow/light fill color of OQ1 in Figure 2.5). Thus, the propositional content of all Communicative Acts assigned to the System implicitly defines the functional interface of the application logic through referring to concepts of the ANM and the DoD models [PR11].

We also support the possibility to provide an additional informal notation of the propositional content (e.g., "Enter username and password" for OQ1). Such shortcut notations have the ad-

vantage that they are more easy to read for human designers and that they can be used as default values for heading labels in the GUI to create.

Overall, our Disourse-based Communication Models are device- and modality-independent and can thus be assigned to the Tasks & Concepts level of the CRF. Another important property of Discourse-based Communication Models is that they represent all possible flows of interaction and thus specify all use cases of a certain application [KPR12].

For more details on Discourse-based Communication Models please see [PR11, Pop12]. The following two subsections are dedicated to two new Discourse Relations (`Ordered Joint` and `Title`) that were introduced to enrich the semantic expressiveness of Discourse-based Communication models for UI generation. Although they are new, we present them here in the Background Chapter, because they thematically belong to the Discourse-based Communication Model.

Ordered Joint

The `OrderedJoint` relation links two or more Nucleus branches and specifies that all branches may be executed concurrently, just like the Joint relation. The difference between these two relations is that the branches of the OrderedJoint are ordered through ascending numbers (starting with 1), in contrast to unnumbered Joint branches. Proving such an order facilitates the layout creation and is beneficial for the usability of the resulting GUI.

The additional information provided through the numbers is exploited during the generation of a GUI in the following way. If all branches are performed in the same presentation unit (i.e., a screen) the order is used to identify the order of presentation (i.e., the position of the branches on a single screen). In the case it is not possible to perform all branches concurrently, i.e., the screen is too small to display all branches together, the relation defines the order for displaying the branches, e.g., the order of screens of a GUI [RPK13b].

The linked Nucleus branches can either contain other Discourse Relations or Adjacency Pairs (as in the running example excerpt shown in Figure 2.5).

The procedural semantics definition of the OrderedJoint is depicted in Figure 2.6 in the form of a UML statemachine. This definition is identical to the one defined for the Joint relation in [Pop12], because the order of concurrent Nuclei cannot be specified in a UML statemachine.

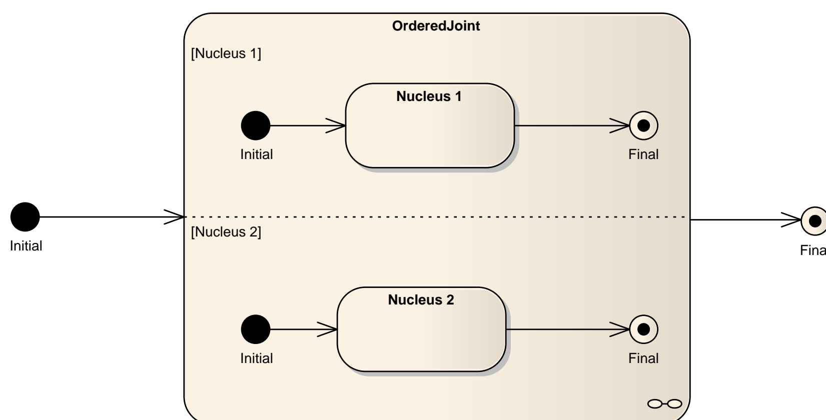


Figure 2.6: Statemachine for (Ordered)Joint

Title Relation

The `Title` relation expresses that the Nucleus branch acts as a title to the Satellite branch. This information is reflected in the layout of the resulting GUI, which places the Nucleus on top of the Satellite by default.

The Title relation is a specialization of the Background relation (for details see [Pop12]), but with different procedural semantics.

The statemachine that defines the procedural semantics for this relation is depicted in Figure 2.7. This definition shows that the outer final state can only be reached via the Satellite branch, which means that the Nucleus branch can be used to display information and does not require any interaction to reach the final state.

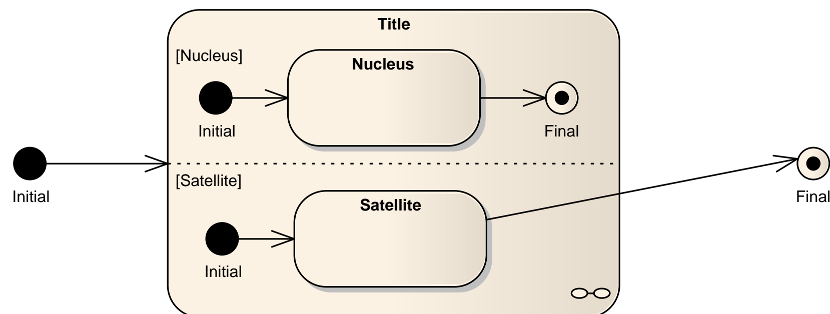


Figure 2.7: Statemachine for Title

Let us illustrate this behavior with the login of our running example (see Figure 2.8). The Satellite branch of the Title relation contains an `OpenQuestion` for the login data (OQ5). The Nucleus branch contains a `Condition`, which means that the `Informing` Communictative Act (I2) is only displayed if the corresponding condition of the `Then` branch (i.e., `loginFailed == true`) is fulfilled at a certain point in time.

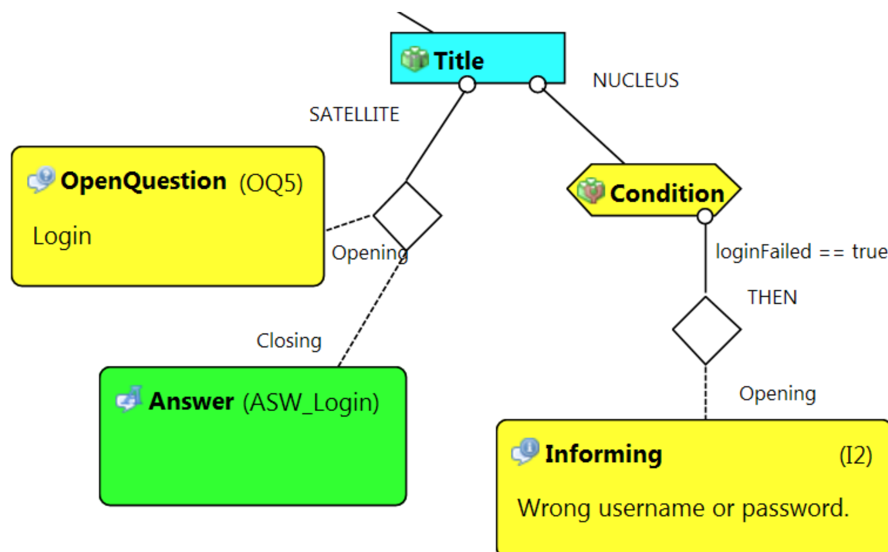


Figure 2.8: Bike Rental Login with Title

2.3.2 Automated User Interface Generation (UCP:UI)

The Unified Communication Platform provides a UI generation framework (UCP:UI) for the automated transformation of Discourse-based Communication models to GUIs, or more precisely: Window / Icon / Menu / Pointing Device (WIMP) UIs.

Figure 2.9 illustrates the basic GUI generation approach of UCP:UI as presented in [KBFK08, Kav11]. The remainder of this section summarizes the main models and transformation concepts involved, because they have been extended to support our new interactive multi-device GUI generation approach presented above (see Figure 1.3).

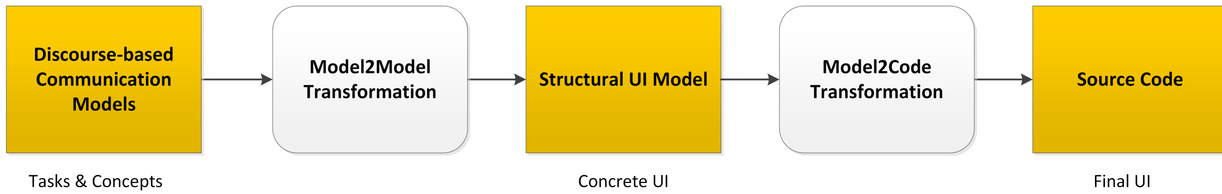


Figure 2.9: Basic UI Generation Approach

2.3.2.1 Structural User Interface Generation

Figure 2.9 shows that model-to-model transformations are applied to transform Discourse-based Communication Models to Structural UI Models. The left-hand side (LHS) of such transformation rules specifies a Communication Model pattern and the right-hand side (RHS) a Structural UI pattern. Structural UI Models provide a device-dependent but still toolkit-independent specification of the GUI and reside on the Concrete UI level of the CRF. This transformation step is completed in two steps [KFK09a], similarly to the concept of lazy or called rules in the Atlas Transformation Language (ATL).

Let us illustrate this two-step process and its implications on the transformation design with an excerpt of the login dialog presented above, where the System asks the User for username and password. Such a login dialog can be modeled as an `OpenQuestion` for the domain concept `User`, with the attributes`username` and `password`. Figure 2.10 shows an example of how such an `OpenQuestion` together with the concept of a `User` can be transformed.

The transformation presented in Figure 2.10 can be performed in one step, if the transformation rule's LHS matches an OpenQuestion with the content User and the RHS creates the corresponding widgets that are required for the specified interaction (i.e., Labels, InputFields and a Button to send the Answer Communicative Act).

The two-step transformation process, however, also allows for the design of more generic transformations. In this case, the first step matches a transformation rule for the OpenQuestion-Answer AdjacencPair with the content of an Object and creates the overall UI structure (i.e., a Panel and Placeholders for the concrete content).

Figure 2.11 shows a transformation rule that matches this LHS pattern. The upper part of this figure shows the source model pattern – an Adjacency Pair with an OpenQuestion and the corresponding answer, where the propositional content of the OpenQuestion is specified as `one EObject`. The lower part shows the corresponding RHS. The RHS is part of the Structural UI Model and consists of the container `Panel`, a heading `Label`, a `Panel` that contains an `Input-` and

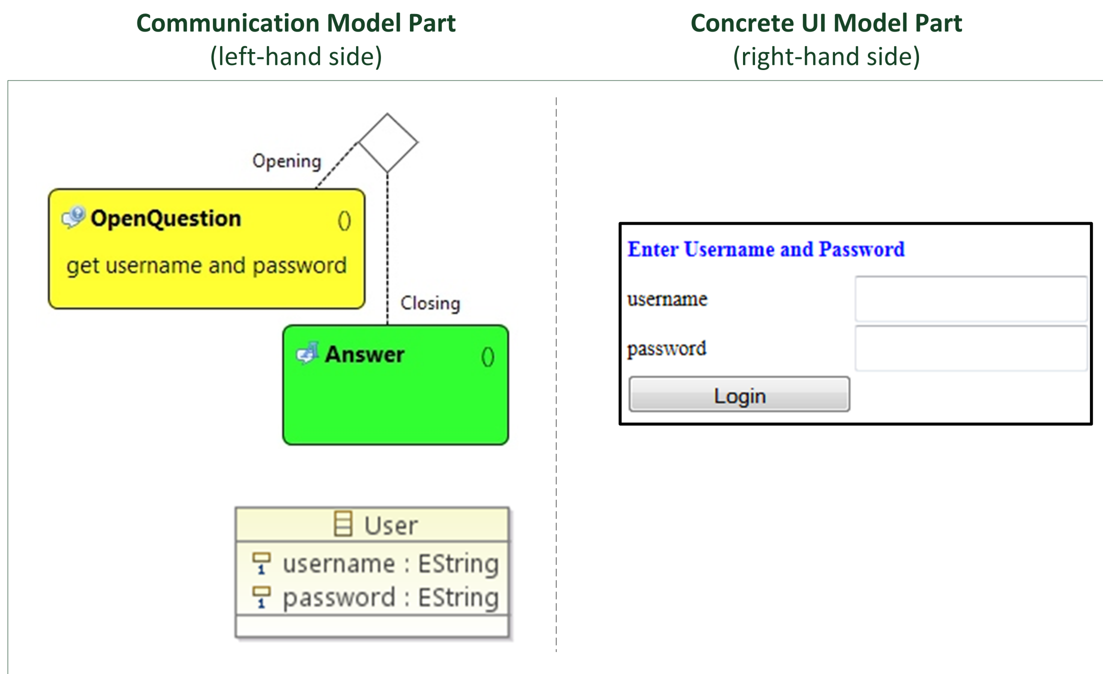


Figure 2.10: Model-to-Model Transformation Example

an `OutputPlaceholder` and a `Button`. This rule is independent of a specific Domain-of-Discourse Model concept, and matches all OpenQuestion-Answer Adjacency Pairs whose propositional content has the type EObject, like the class User from our login example.

The second step iterates over all attributes of the `EObject` and calls a second-level rule that generates a concrete interaction widget according to the type of the attribute. In our example, there are two attributes: username and password, both of type `EString`. Therefore, the `OutputPlaceholder` is replaced through a `Label` for each attribute name and the `InputPlaceholder` through a `Textbox` for each attribute, which allows the user to insert the requested information (as illustrated at the right side of Figure 2.10).

This approach already supported the tailoring of the transformation according to pointing granularity (i.e., mouse/pen vs. finger-based operation) [KRF+09]. The pointing granularity was specified in a so-called Application-tailored Device-specification and used to filter the set of transformation rules subsequently. The concept of the Application-tailored Device-specification was extended to support different scrolling options int the context of automated GUI optimization [RPK+11b] and is presented in detail in Section 4.1.

For details on the transformation process please see [KFK09b, Kav11]. More detailed information on the design of generic transformation rules is presented in Section 4.2.

2.3.2.2 Behavioral User Interface Generation

A behavioral UI specification is required apart from a structural UI specification to completely define a UI. In UCP, the behavior of the UI is specified through the flow of interaction modeled in the Communication Model. The behavior of all Communication Model constructs (i.e., Adjacency Pairs and Discourse Relations) is defined through UML statemachines and the overall UI behavior

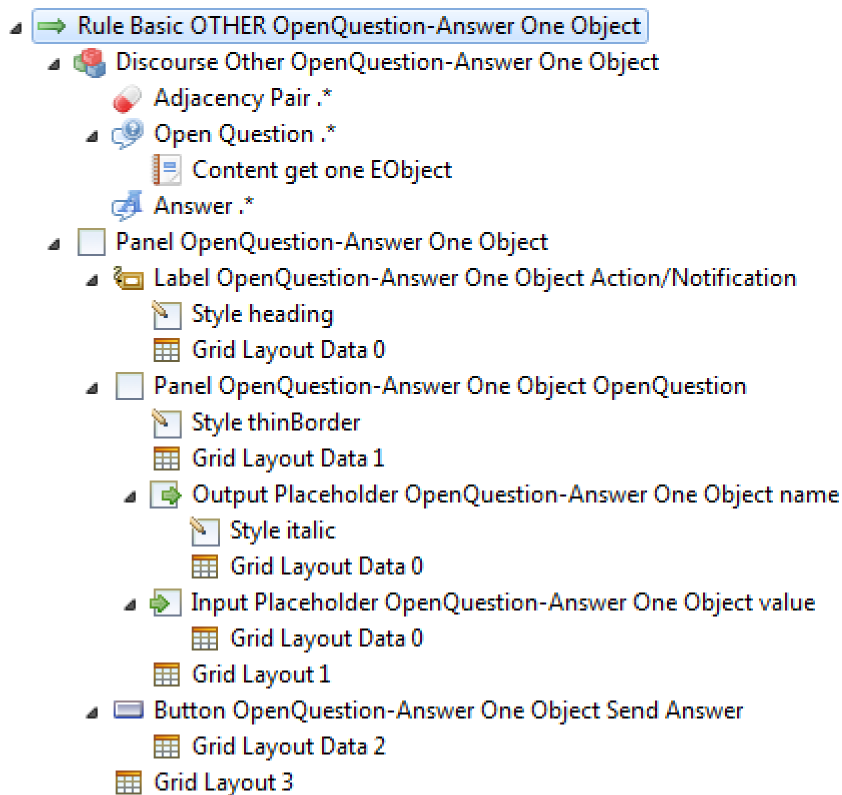


Figure 2.11: OpenQuestion-Answer Rule for one EObject

is thus defined through the composite statemachine of a certain Communication Model. The creation of this composite statemachine together with a basic algorithm to derive a UI Behavior Model from this composite statemachine has been presented in [PFA+09] and is summarized here based on [RPKF11].

A UI Behavior Model is a state-transition network with so-called *Presentation Units* (PUs) as states. The minimum information unit granularity of this statemachine are Communicative Acts, which are also the triggers for its transitions. Each PU is defined through a unique set of Communicative Act instances, specified in the corresponding Discourse-based Communication Model. This set consists of all Communicative Acts that have been *received* by the user (i.e., uttered by the system) and all Communicative Acts that can be *sent* (i.e., uttered) by the user via the UI at a particular point in time. The UI Behavior Model is derived from the device-independent Communication Model, without the consideration of any device-specific characteristics. This keeps the UI Behavior Model device-independent and means, e.g., that a PU does not take the screen size of the target device into account. It rather assumes that all information fits on a (potentially infinite) screen.

The Presentation Unit Derivation algorithm presented in [PFA+09] extracts the PUs in two steps. The first step requires that a UML statemachine is given for each Discourse Model element, which defines its procedural semantics. It combines these statemachines into a composite statemachine according to the Discourse Model hierarchy.

In the second step, the algorithm analyzes this composite statemachine to derive the PUs, based on the assumption that the machine utters all Communicative Acts that are possible at a certain

point in time according to the Discourse Model. The algorithm begins at the statemachine's start node and searches all outgoing branches until it finds a Communicative Act which the user has to send via the UI. While traversing the statemachine, it collects all Communicative Acts uttered by the system (i.e., received by the user) and the first Communicative Act that the user sends for each branch. Therefore, the resulting set of Communicative Acts for the corresponding PU consists of the received Communicative Acts uttered by the system, and the sent Communicative Acts the user can utter via the UI. Subsequently, the algorithm continues to traverse the statemachine from the states where it stopped to extract the next PU. The Communicative Acts that are exchanged between neighboring PUs (i.e., Sent Communicative Acts from the source PU and Received Communicative Acts of the target PU) are the triggers for the corresponding transitions. The algorithm continues until all branches in the composite statemachine have been analyzed. This algorithm is illustrated with a small example in Chapter 3, together with related problems in the context of GUI generation.

2.3.2.3 Source Code Generation

The second transformation step of the basic transformation approach is a model-to-code transformation, which generates the corresponding source code for a given Structural UI Model. This transformation is based on Java Emitter Templates (JET) and has been implemented in the past for Java Swing [Ran08], but it is currently provided for HTML only.

2.3.3 Runtime Platform (UCP:RT)

The Unified Communication Platform contains a run-time module (UCP:RT), which provides a run-time architecture based on the Model-View-Controller (MVC) pattern to integrate the generated GUI with the application back-end [Pop12]. This software module also supports the composition of existing UCP:MachineServices to new ones and can thus be used as a Service-oriented Architecture (SOA) platform as well [Pop12].

3 Automated Screen Model Generation

This chapter shows how we extended the basic UCP:UI transformation approach to provide a screen-based GUI model on the Concrete UI Level as a basis for manual customization through the designer. The upper part of Figure 3.1 shows the Basic GUI Generation approach presented above, and the lower part depicts the Interactive GUI Generation approach as proposed in this doctoral dissertation [Ran10]. The interactive approach extends the basic approach with an additional "Structural UI to Screen Model Conversion" step, depicted as rounded rectangle, resulting in the Screen Model, depicted as yellow rectangle.

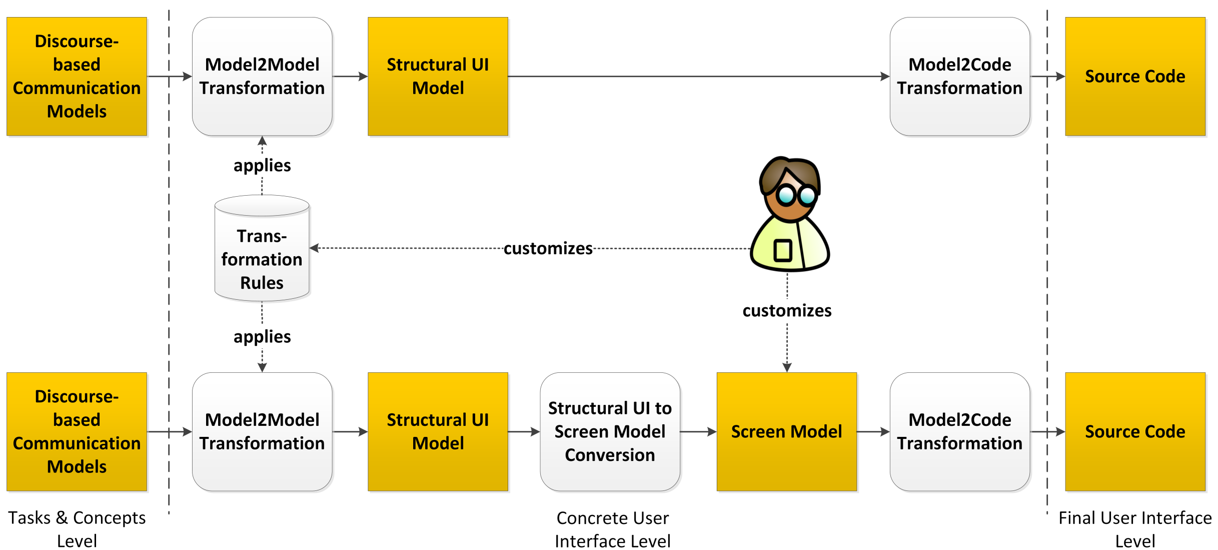


Figure 3.1: Automated Screen Model Generation based on [Ran10]

In particular, we show how the composite statemachine that represents the UI Behavior Model is transformed into a WIMP-UI specific statemachine and how this statemachine is weaved together with the Structural UI Model to provide a (structural and behavioral) Screen Model. Both steps are performed fully automatically. An additional benefit of the Screen Model is that it allows for automated tailoring in the context of multi-device GUI generation, as shown in the next chapter.

3.1 WIMP-UI Behavior Generation

This section presents our approach to automatically transforming the UI Behavior model, presented above, into a so-called WIMP-UI Behavior Model based on [RPKF11].

We claim that a device-independent high-level specification should allow expressing parallelism whenever feasible. The parallel information can then be serialized according to the properties of the target device (i.e., modality and platform) during the rendering process. In contrast to, e.g., speech input and output, WIMP-UIs can present and handle parallel communication units within the limits of the given display size and resolution. So, it is sometimes possible to transfer such parallelism directly. If this property of WIMP-UIs is not taken into account adequately during the generation process, certain usability problems of the generated WIMP-UIs arise, which are illustrated below.

3.1.1 Device-independent WIMP-UI Behavior Generation

This subsection introduces our approach for automatically generating a UI behavior model that allows for handling parallel communication units, based on the Presentation Unit (PU) derivation algorithm presented in [Pop09] and summarized in Subsection 2.3.2.2. Sending more than one Communicative Act at the same time was out of scope for this algorithm. Therefore, it extracts a new PU with each user-sent Communicative Act. This is perfectly suited for serial communication channels (e.g., modalities like speech or machine-machine communication) and the algorithm even derives correct PUs in the case of WIMP-UI generation, but the granularity of the possible answers is not well suited for WIMP-UIs, especially for ones with a larger screen. This granularity mismatch means that the algorithm forces the user to serialize the information (i.e., send one Communicative Act after the other) instead of exploiting the inherent strength of a WIMP-UI to send information in parallel. Thus, the UI Behavior Model requires more button clicks than necessary and contains more PUs (i.e., screens) than necessary. Both issues impair the usability of the final GUI.

Let us illustrate these issues with a simplified flight selection scenario (without information on flight date etc. involved). In a first step, the user selects both departure and destination airports for the flight. Subsequently, she chooses one flight from a list offered by the booking system. Based on these data, the system checks whether there are seats available on this flight and either lets the user select a seat or informs her that no seats are available (i.e., the flight is already overbooked).

The corresponding Discourse Model is illustrated in Figure 3.2. This model contains a `Joint` relation that links the two ClosedQuestions for the departure and destination airport selection. The Joint relation models concurrency and has the same procedural definition as the OrderedJoint relation presented above. The additionally used `Condition` relation is assigned to the System, which means that the System selects either the Then or Else branch, based on the evaluation of the Then branch condition (i.e, `seats available == true`).

The excerpt of the UI Behavior model that has been derived for the Joint subtree of this Discourse, using the basic Presentation Unit derivation algorithm [Pop09] sketched above, is presented in Figure 3.3. Each rounded box represents a state which corresponds to a PU. The top compartment shows the name of the PU. The middle compartment shows the sub-set of Communicative Acts received by the user while transitioning from a previous PU. The bottom compartment specifies

the sub-set of Communicative Acts that can be sent by the user while in this PU. The transitions are labeled with the ID of the Communicative Act sent by the user, followed by the IDs of the Communicative Acts to be received in the next PU.

`PU1 Airport Selection` represents the screen in which the user can select the departure and the destination airports. The outgoing transitions of `PU1 Airport Selection`, however, have only one Communicative Act as a trigger each. This means that the user can either send the departure airport (which would lead to `PU3 Destination Selection`) or the destination airport (this would lead to `PU2 Departure Selection`) as an answer. From the next PU, she can send the other airport selection and the machine will offer her a list of available flights (`PU4 Flight Selection`).

A possible final UI for `PU1 Airport Selection` is shown in Figure 3.4. This screen displays the information of the two Communicative Acts `select departure airport (1)` and `select destination airport (3)` to the user and offers the possibility to send one of the two corresponding answers (2 or 4).

The screen in Figure 3.4 represents a situation in which the user would like to send both answers at the same time (as foreseen by the *Joint* relation in Figure 3.2). The Presentation Unit Derivation algorithm, however, assumes that the user sends only one answer at any time. This leads to two additional PUs in our Flight Selection example.

The serialized UI Behavior Model results in two major usability problems. First, it leads to a confusing GUI with a 'Submit' Button for each answer that can be sent by the user (see Figure 3.4). The second problem regards confusing screen changes during which the already answered subpart 'disappears'. This means that the drop-down menu for the departure airport selection in Figure 3.4 disappears due to a screen change, as soon as the user clicks the upper submit button. Another drawback related to this problem is that the algorithm leads to a large number of PUs, most of which are actually not needed in a WIMP-UI.

Regarding our Flight Selection example, removing Presentation Unit `PU2 Departure selection` and `PU3 Destination selection`, highlighted in Figure 3.3 with a dotted rounded box, and combining the corresponding transitions into one direct transition from `PU1 Airport Selection` to `PU4 Flight Selection` with both answers as triggers would be possible and desired.

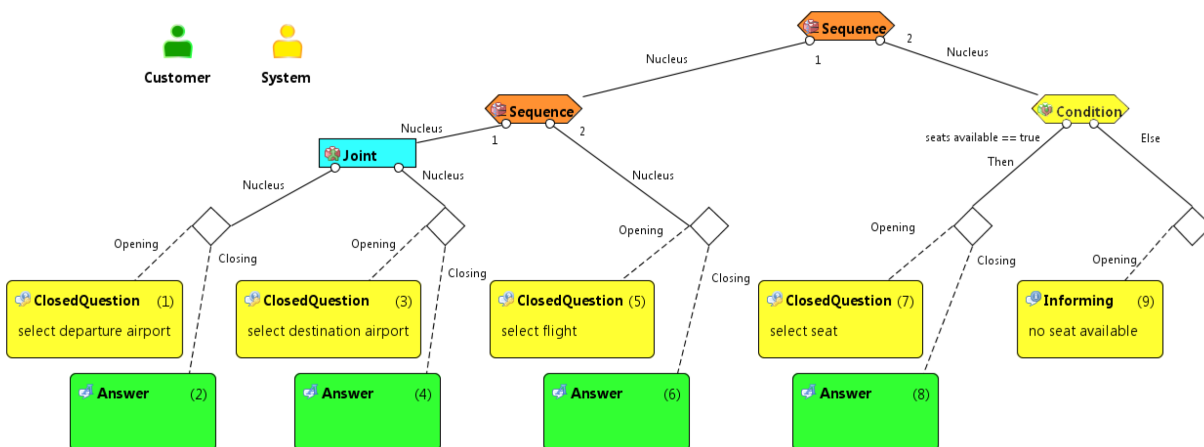


Figure 3.2: Flight Selection Discourse Model [RPKF11]

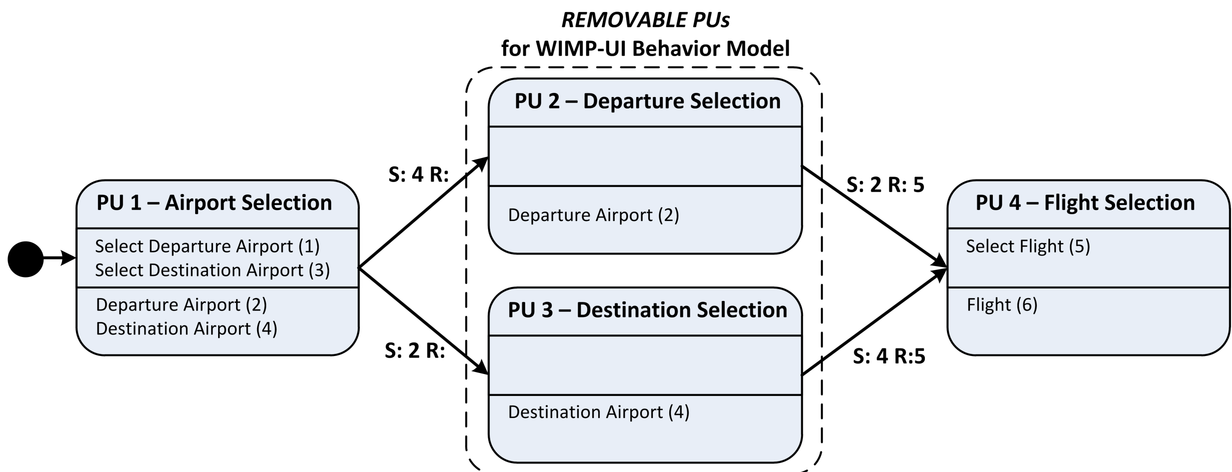


Figure 3.3: UI Behavior Model for Joint Subtree [RPKF11]

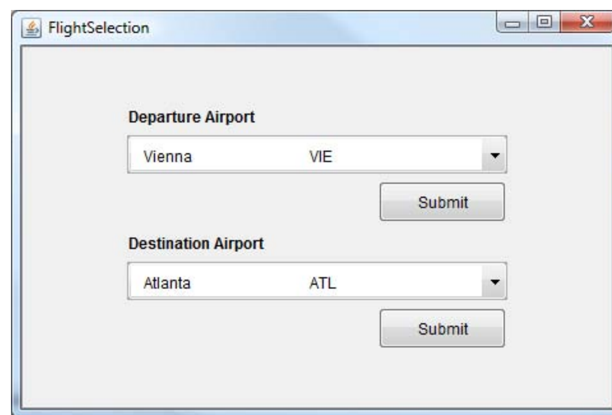


Figure 3.4: Flight Selection WIMP-UI with Problems [RPKF11]

To avoid these usability problems, we extended the existing approach with a new step that adapts the granularity of communication units as illustrated in Figure 3.5. This *Granularity Adaptation* shifts the granularity of answers from one Communicative Act to a set of Communicative Acts, which contains all answers that the user can send at once in a given PU.

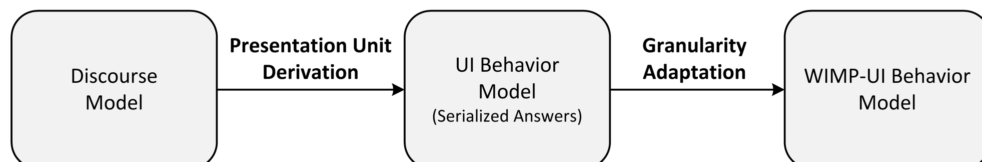


Figure 3.5: WIMP-UI Behavior Generation [RPKF11]

The modifications for adapting granularity consist primarily of the elimination of PUs whose set of Communicative Acts is a subset of a previous PU and the adjustment of the corresponding transitions.

Let us illustrate these adaptations with our running example. The UI Behavior Model in Figure 3.3 is the result of the first transformation step. The subsequent granularity adaptation analyzes all PUs that can be reached through Communicative Acts sent by the user. Our new

algorithm checks whether there is a PU without received Communicative Acts (empty top compartment), and all sendable Communicative Acts (displayed in the lower compartment) are a subset of the current PU's sendable Communicative Acts. This is the case in `PU2 Destination Selection` and `PU3 Departure Selection`, whose Communicative Acts build a subset of `PU1 Airport Selection`'s Communicative Acts. The new granularity adaptation step removes these PUs and combines their incoming with their outgoing transitions. The trigger of the new transition is made up of the triggers of the transitions to merge. First, we remove `PU2 Departure Selection` and merge its transitions into one transition from `PU1 Airport Selection` to `PU4 Flight Selection` with the triggering Communicative Acts `2` (departure airport) and `4` (destination airport). Next we remove `PU3 Destination Selection`. We just remove this PU and its incoming and outgoing transitions, because a transition from `PU1 Airport Selection` to `PU4 Flight Selection` with the triggering Communicative Acts `2` and `4` already exists.

Figure 3.6 shows the complete WIMP-UI Behavior Model that we have generated in this way for the Flight Selection example. The WIMP-UI Behavior Model with the adapted granularity is shown in the dotted rounded box labeled *JOINT*. This figure additionally shows the PUs corresponding to the *Condition* relation of Figure 3.2. Both outgoing transitions of `PU4 Flight Selection` specify Communicative Act `6` (the flight selected by the user) as the user's sent Communicative Act set of the trigger. This means that the machine decides which transition is actually triggered. Depending on the machine's decision, the user shall receive either Communicative Act `7` or `9`. This leads then to either `PU5 Seat Selection` or to `PU6 No Seat Available`.

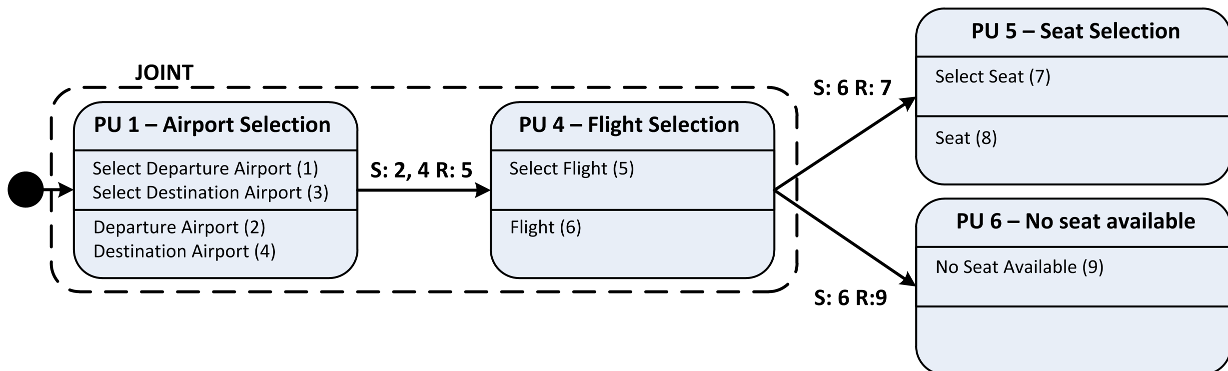


Figure 3.6: WIMP-UI Behavior Model for the Flight Selection Discourse [RPKF11]

In effect, our new transformation approach creates a *WIMP-UI Behavior Model* at Presentation Unit level (i.e., maximum available information presented in parallel). Compared to the previous Presentation Unit Derivation algorithm, it usually results in fewer PUs, and the user can trigger the transitions between the PUs by a set of 1 to n Communicative Acts instead of just a single Communicative Act.

Figure 3.7 shows a possible Final UI for `PU1 Airport Selection` in Figure 3.6. In contrast to Figure 3.4, there is only one "Submit" Button that sends both selected airports at once. Clicking this button directly leads to the flight selection screen corresponding to `PU4 Flight Selection` in Figure 3.6.

It turned out while implementing this algorithm that it is more efficient to assemble the composite statemachine and immediately analyze it incrementally in each Discourse Model node, compared to assembling the statemachine for the whole Discourse Model before analyzing it. This allows merging transitions that are triggered by Communicative Acts sent by the machine and thus,

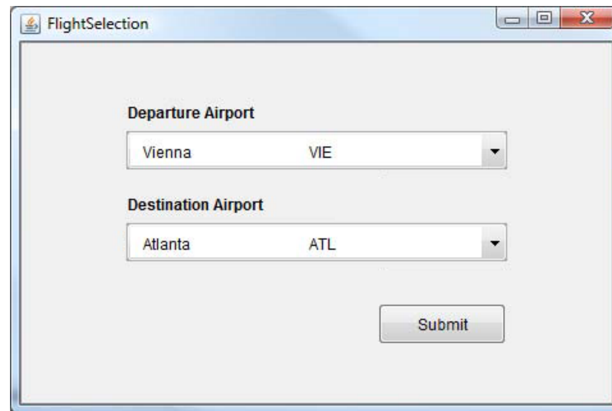


Figure 3.7: WIMP-UI Behavior Model for the Flight Selection Discourse [RPKF11]

to reduce the number of PUs and transitions in each node. As a consequence, the number of branches that need to be analyzed in the next node is reduced.

Primarily, this granularity adaptation supports sending more than one answer in parallel, which avoids the related usability problems. Another benefit of our new algorithm is the reduced number of PUs. This leads to less effort for manual UI customization by a designer through a reduced number of screens, and finally to less source code.

Table 3.1 provides the numbers of derived PUs before and after the granularity adaptation algorithm for different Discourse Models. The first example is the Flight Selection Discourse Model that we used to illustrate our approach. The Flight Booking Discourse Model is a more elaborate version where a user can book and pay a flight ticket, and the OnlineShop Discourse Model is a small e-commerce example. The variation in the differences between the numbers of screens before and after the granularity adaptation as illustrated in Table 3.1 is due to the different number of relations that allow parallelism in each Discourse Model.

Table 3.1: Numbers of screens for different Discourse Models [RPKF11]

Discourse Model	Number of Screens *BEFORE* Granularity Adaptation	Number of Screens *AFTER* Granularity Adaptation
FlightSelection	6	4
FlightBooking	25	7
OnlineShop	13	7

Our WIMP-UI Behavior model generation process does not take the actual screen size of the target device into account and results, therefore, in a target device-independent WIMP-UI behavior model. This model is needed as a basis for the subsequent transformation step, presented in the next section, which tailors it to a certain device.

3.1.2 Beyond the State-of-the-Art

This subsection presents dynamic UI model development in state-of-the-art UI generation approaches and highlights similarities and differences to our approach.

The MARIA Environment [PSS09b] offers tool support for the transformation of CTT models into UIs. The algorithm presented in [PBSK99] analyzes the hierarchy and the temporal operators

of a task model to derive so-called Presentation Task Sets (PTS). A PTS contains all tasks that are enabled at the same time. Each PTS is enriched with one or more navigators that point to the next PTS according to the task tree on Abstract UI level. Because CTT applications are typically user-driven, the approach assumes that each decision is made by the user. This implies that incorrect PTSs and subsequently navigators are derived. This can be illustrated with our small Flight Selection scenario, where MARIAE's algorithm derives two explicit navigators for the "Flight Selection" presentation unit (PU4 in Figure 3.7). One of these navigators leads to the "Seat Selection" presentation unit (PU5 in Figure 3.7) and the second one to the "No seat available" presentation unit (PU6 in Figure 3.7), which gives the Customer the choice how to proceed instead of the System (as modeled in our running example).

Another algorithm that transforms task models into so-called Activity Chains [LCCV03] has been developed and implemented as part of the Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems (Dygimes) [CLV+03] framework. This approach formalizes an Activity Chain as a state-transition network. The states represent Enabled Task Sets (ETS), which are related with transitions that are derived from the temporal operators in the task model. Task models, however, do not specify the events that are exchanged between the user and the system at run-time, they only model which tasks have to be performed by the user. This means that the trigger events for the transitions cannot be derived automatically from the task model, but have to be added manually by the designer.

An approach that transforms task models into hierarchical Unified Modeling Language (UML) statemachines is presented in [vC07]. This approach maps each temporal relation of the CTT model to a UML statemachine and uses guard conditions for the transitions to model the relation's characteristics. This approach claims to have greater expressiveness compared to the Activity Chain approach sketched above, through the support of nested states. Compared to our approach it uses guard conditions instead of events. Hence, this approach can also not derive the triggers for the transitions automatically.

A model-driven GUI generation approach for information-driven applications is presented in [PEPA08]. This approach combines task models and GUI sketches to derive a presentation model. The dynamic model, however, is created manually.

Another UI generation approach that derives WIMP-UIs from task models (using CTT notation) is presented in [WF09]. This approach was extended with so-called *markers* to annotate the tasks, as task models themselves are not sufficient to express the dynamic relations within a UI [WF10]. A task's *marker* specifies how its subtasks are grouped into views. Subsequently the *Dialog Graph*, which explicitly specifies the navigation structure of the WIMP-UI, can be derived automatically by exploiting the markers.

Compared to our approach, the latter two approaches use manual creation or annotation and, therefore, do not have to deal with problems related to automated behavior model generation. The other approaches sketched above support the automated derivation of a UI behavior model, but they do not support automated derivation of the trigger events for the corresponding transitions. In contrast to these approaches, we support the fully automatic generation of a state-transition network, the WIMP-UI Behavior Model, out of a high-level Discourse Model, including the derivation of the run-time trigger events for the transitions.

3.2 Screen Model Generation

This section presents our approach for *weaving* a device-dependent Structural UI Model and a device-independent WIMP-UI Behavior Model to derive a screen-based UI model on the Concrete UI level, as presented in [RPK+11a]. Weaving means here to associate elements of two models that describe the same UI from different points of view to create one model that represents both views. In effect, the resulting *Screen Model* consists of both a device-specific Structural Screen Model and a related Behavioral Screen Model. That is, the states of this behavioral model represent the screens of this structural model. Such a Screen Model can be customized manually by a designer for improving style and layout, and from such a screen model the final WIMP UI can be created in a straight-forward manner.

Let us again introduce a small flight booking example to illustrate the weaving process. This scenario differs slightly from the one used above, to better fit our needs for illustrating the weaving process. In the scenario used here, the user selects the departure and destination airports. Subsequently, the system informs about the two selected airports. The corresponding Discourse Model is shown in Figure 3.8.

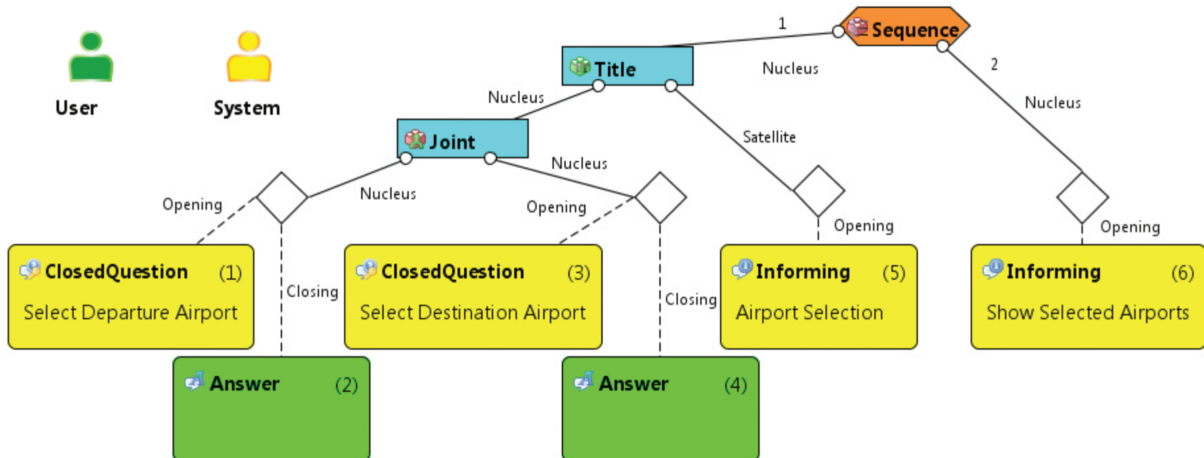


Figure 3.8: Airport Selection Discourse Model

Figure 3.9 depicts the Structural UI Model corresponding to our running example. The top `Choice` element contains two `Panels` representing both children of the Sequence relation shown in Figure 3.8. Since we rendered the running example for a very small screen (220×176 pixels), the transformation selects a rule for the Joint relation in Figure 3.8 that splits the information of both branches into two panels that can be displayed alternatively (indicated by the embedded Choice element). Each panel consists of a `List Widget` for selecting an airport that will be rendered as a combo box and a submit button.

Figure 3.10 shows the generated UI Behavior Model for our running example. The first Presentation Unit (*PU1 – Airport Selection*) displays the Informing *Airport Selection* and the two ClosedQuestions for the airport selection. It allows the user to send the corresponding answers. The second Presentation Unit (*PU2 – Selected Airports*) presents the selected airports to the user. The Answer Communicative Acts (2 and 4) sent by the user at once trigger the transition between the two Presentation Units.

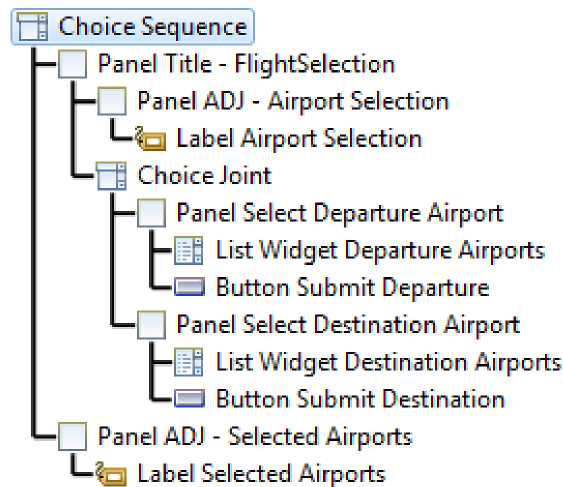


Figure 3.9: Device Specific Structural UI Model

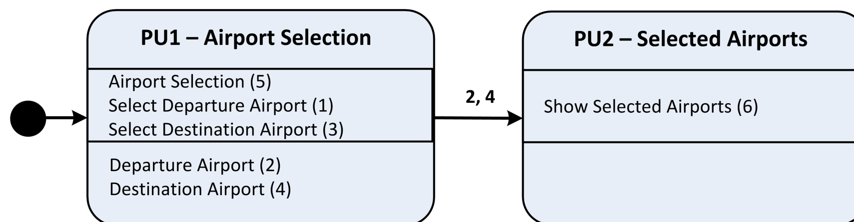


Figure 3.10: Device Independent UI Behavior Model

3.2.1 Weaving of Structural and Behavioral Models

Figure 3.11 details the interactive GUI generation approach presented above in Figure 3.1 and shows that a Communication Model is transformed into a WIMP-UI Behavior Model and a Structural UI Model, under the consideration of a Device Specification. Subsequently, the device-*independent* UI Behavior Model and the device-*dependent* Structural UI Model are weaved for generating the Screen Model. This resulting model defines both the structure of the WIMP UI and its behavior. It provides the basis for the code generation.

The Screen Model actually consists of the *Structural Screen Model* and the *Behavior Screen Model*. The Structural Screen Model for our running example is shown in Figure 3.12 and specifies the concrete screens of the WIMP UI. The corresponding Behavior Screen Model is shown in Figure 3.13 and specifies the behavior of the WIMP UI, i.e., the possible sequences of screens.

The new weaving process has the following five steps:

1. Analyze Structural UI Model and create a list of panels that belong to a screen.
2. Extract (received/sent) Communicative Acts for each screen.
3. Analyze WIMP-UI behavior model and create a screen for each PU.
4. Merge buttons to send more than one Communicative Act at once.
5. Split PUs (if required) according to Structural Screen Model.

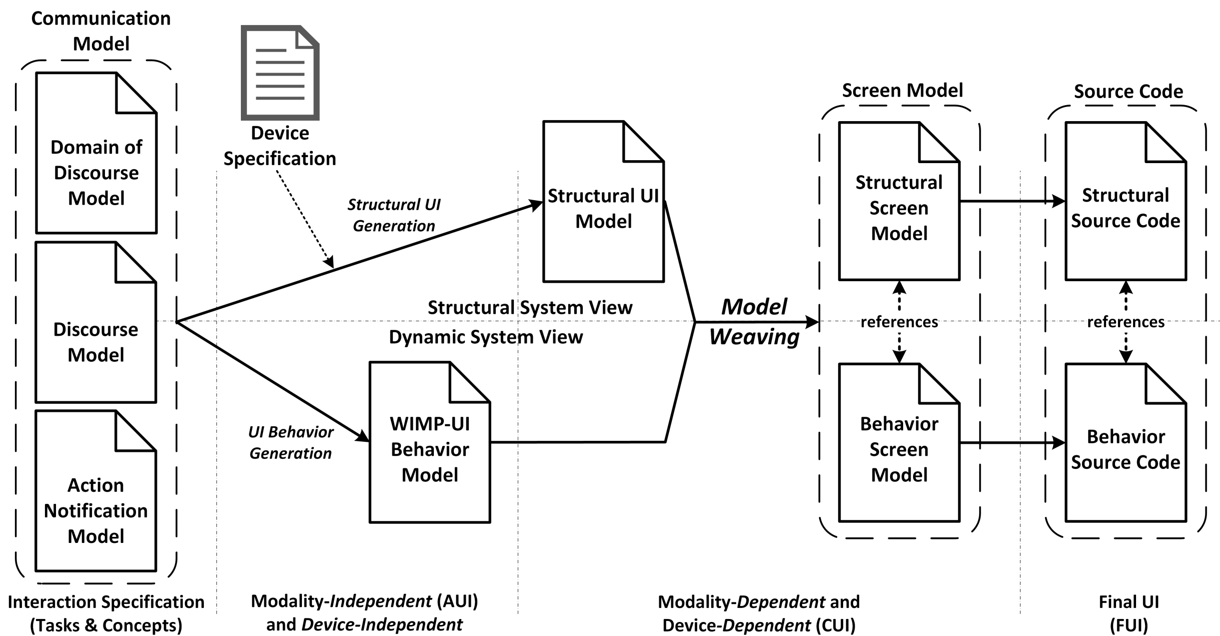


Figure 3.11: Interactive WIMP-UI Generation Process [RPK+11a]

The *first step* traverses the Structural UI Model top-down and creates a list of Panels that belong to a screen. The list of Panels is gradually refined according to the containers in the Structural UI Model. A *Panel* means that its Sub-Panels are added to the same screen, whereas a *Choice* element means that the screen is split before the children are added.

The *second step* uses the traces between the Structural UI widgets and the Discourse Model elements to create a map that assigns the Communicative Acts that are displayed or can be sent to each screen. Table 3.2 shows this map for our running example.

Table 3.2: Map Communicative Acts from Figure 3.8 to Screens

Communicative Acts	Panels in Screen
Informing_5, ClosedQuestion_1, Answer_2	Title – FlightSelection, ADJ – Airport Selection, Select Departure Airport
Informing_5, ClosedQuestion_3, Answer_4	Title – FlightSelection, ADJ – Airport Selection, Select Destination Airport
Informing_6	ADJ – Selected Airports

Subsequently, this map and the UI Behavior Model are used to create the Structural Screen Model in the *third step*. Each screen consists of the Panels defined in the map. Some Panels, like the `Title -- FlightSelection` Panel in our example, may belong to more than one screen. Such Panels are added to the corresponding screens in the Structural Screen Model.

The *fourth step* checks whether each Presentation Unit can be mapped directly to a screen, or if a Presentation Unit consists of several sub-screens. No adjustments on the Presentation Units are needed in case of direct mapping, because each Presentation Unit corresponds to a screen. In this case the algorithm analyzes which information (i.e., Communicative Acts) is required to trigger a certain transition and combines the corresponding buttons. The other case means that screens have been split during the Discourse Model to Structural UI Model transformation. This

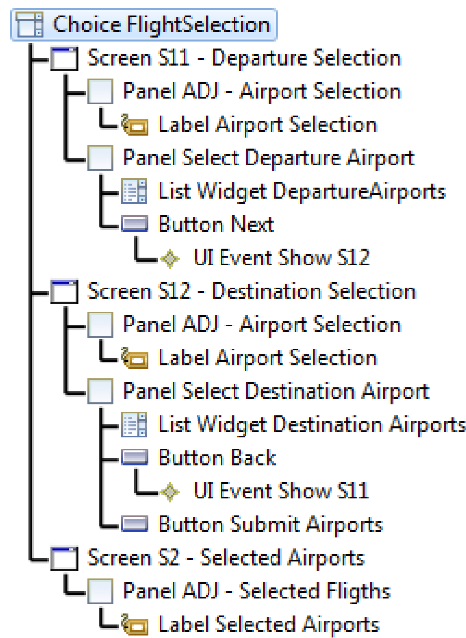


Figure 3.12: *Structural* Screen Model

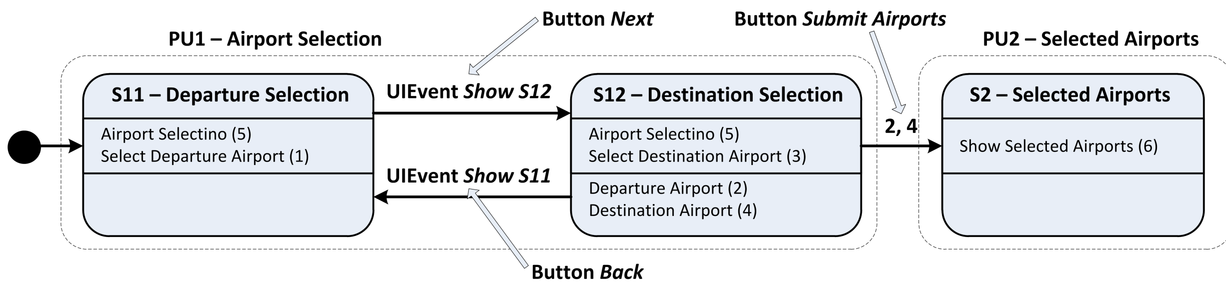


Figure 3.13: *Behavior* Screen Model

implies that `Buttons` need to be added that trigger screen changes between the screens that have been split in the Structural UI Model according to device constraints. We introduce so-called *UIEvents* to model those events that are only specific to the WIMP UI but not defined in the high-level Discourse Model.

In the Structural Screen Model of our running example shown in Figure 3.12, the *Next* Button in Screen *S11 - Departure Selection* and the *Back* Button in Screen *S12 - Destination Selection* trigger the UIEvents for the corresponding screen changes. Additionally, the Buttons that send the Communicative Acts are combined to one Button that sends all Communicative Acts concurrently in the last screen. Which Buttons can be combined is derived from the triggers in the Screen Behavior Model. In our example, this is the *Submit Airports* Button in Screen *S12 - Destination Selection*.

The *fifth step* is the transformation of the WIMP-UI Behavior Screen Model to the Behavior Screen Model. The split screens are added to this model as new states that replace the corresponding original Presentation Unit. Figure 3.13 shows the resulting model for our running example.

For this example, we chose a target device and a platform that require explicit screen splitting

(e.g., a Smartphone with pure HTML support). Alternatively, if the target platform with the same screen size supported Tabbed Panes (e.g., technology like Java Swing or Java Script for HTML), such a tabbed pane could be used as a container for the two screens in our example. In this case, the screen-splitting would be handled by the Tabbed Pane widget and, therefore, no adaptations in the Behavior Screen Model would be necessary. The introduction of UIEvents allows us to support even less powerful target platforms (e.g., Smartphones without Java Script support).

The Screen Model provides a consistent specification of the WIMP UI. The relation between the Structural Screen Model and the Behavior Screen Model is represented by the trigger events. This implies that each state in the Behavior Screen Model corresponds to exactly one screen in the Structural Screen Model and vice versa. Thus, the Screen Model provides a complete and consistent definition of the screens and their possible sequences at compile time. It can be transformed to source code in a straight-forward manner, without any model analysis at run-time. This is especially important for devices with limited capacities (e.g., embedded devices). Moreover, the Screen Model provides a complete WIMP-UI specification at Concrete UI Level and, therefore, a sound basis for manual customization by the designer.

3.2.2 Beyond the State-of-the-Art

This subsection presents state-of-the-art UI generation approaches and highlights similarities and differences to our approach.

The Dygimes Framework [CLV+03] combines task specifications with XML-based UI building blocks to generate UIs for mobile computing devices and embedded systems. Such UIs can adapt to the context of use at run-time and thus migrate over devices. Which tasks are enabled in the same presentation is defined by the UI building blocks that need to be provided by the designer.

GrafiXML [MV08] is an intelligent UI builder that supports the development of UIs for multiple contexts of use (i.e., many users, platforms and environments). The UIs are developed manually in a graphical editor and assigned to a certain context of use. GrafiXML uses UsiXML[1] as its UI specification language. This adds additional power as its UIs can be manipulated or refined in other UsiXML tools as well.

A top-down generation approach that supports the transformation of ConcurTaskTrees (CTT) into UIs for multiple devices is presented in [MPS04]. This approach uses a common task model for all UIs, which is subsequently refined manually by the designer to the System Task model. The System task model is already tailored to the target device and defines which tasks are concurrently enabled and thus belong to the same presentation. System task models are the starting point for the automated UI generation process.

Another CTT-based approach is introduced in [PSS09a], which supports the creation of UIs based on pre-existing functionalities in the form of Web services. This approach supports UI refinement in terms of interaction techniques (i.e., modalities or widget selection). Which UI elements are available at a given time and thus belong to the same presentation, is still defined by a CTT model.

All these approaches use task models on their highest level of abstraction and provide models on all levels of the Cameleon Reference Framework during the UI generation. Our approach does not

[1] http://www.usixml.org

provide a Structural UI Model on AUI level, because automatically tailoring the GUI model for a given display size requires to know the size and layout of the resulting GUI, which is available at CUI level. Skipping the AUI level on the Structural side, but keeping it on the Behavioral side, to support multi-modal UI generation in principle, requires the weaving of structural and behavioral models that we presented in Figure 3.11.

Our approach is able to fully automatically generate device-specific WIMP UIs out of the same device-independent Communication Model (based on discourses). All the task-based approaches above, in contrast, need one or the other manual intervention by a human designer (e.g., through adding device-specific annotations).

4 (Semi-)Automatic GUI Tailoring for Multiple Devices

Considering device characteristics during the GUI design, i.e., tailoring the GUI for a specific device, is important for achieving a good level of usability. Automating this process potentially saves development time and effort. This benefit is even greater if touch-based devices are involved, because such devices can be operated in portrait or landscape mode and thus potentially require two GUIs, tailored to the respective mode as well.

This chapter presents our approach for providing the designer with a GUI, tailored for a certain device according to given optimization objectives, together with our concepts on how to involve the designer in this process. Additional input from the designer is possible and in general required to achieve a satisfactory level of usability for the resulting GUI, but not mandatory in our approach. For this reason, we put the "semi" in the chapter title in parentheses. We start with the presentation of how and, in particular, which aspects of a platform in the sense of target-device we model. Subsequently, we present a *basic* transformation rule set that we devised and implemented. This rule set has two important features. First, it is complete in the sense that every compliant Communication Model can be transformed. Compliant means that it is compliant to its meta-model and to additional constraints, which we specified using OCL and Java [RSKF11]. Second, it allows for the generation of GUI alternatives and thus for selecting an optimal GUI according to given optimization objectives.

Our transformation rules create parts of the Structural UI Model, which are structured hierarchically in the form of a tree, corresponding to the structure of the Discourse Model tree. Each screen contains one or typically more such Structural UI Model parts that are combined to screens during the weaving process with the WIMP-UI Behavior Model and placed in their container with a "valid" layout to fully specify the Structural Screen Model. With valid in the context of layout we mean that it is complete (i.e., fully specified in the sense that no layout information is missing) and technically correct (i.e., directly matchable to layout specifications in specific GUI toolkits like Java Swing or HTML). The third section of this chapter presents our automated layout approach, which can be influenced by the designer through providing so-called "Layout Hints". In the last section, we present our approach for device-tailoring through automated GUI optimization and four different strategies for tailoring a GUI automatically, which can be selected and applied by the designer, based on this approach. This facilitates the exploration of rendering alternatives. Moreover, we present the interplay between transformation rules, layout module and tailoring strategies. Overall, this chapter focuses on the technical details that are required by the development process for interactive applications, presented in Chapter 5.

4.1 Application Tailored Device Specification

A platform in the context of UCP is defined through hardware, software and non-functional characteristics. Non-functional characteristics specify how this device is to be used by a certain application (i.e., finger- or pen-based interaction on a touch screen and whether scrolling is allowed up to a certain length/width). All these properties can be specified in the form of a so-called *"application tailored device specification"*. The currently used specification is based on the one introduced by Kavaldjian et al. in [KRF+09, Kav11] and specifies the properties shown in Table 4.1.

Table 4.1: Application Tailored Device Specification Properties

name	defines the name of a certain device.
resolution	specifies the x and y resolution of the device's display.
dpi	specifies the dots per inch of the device's display.
defaultCSS	specifies the default CSS to be used for the device (e.g., to specify the minimum size of a button).
pointingGranularity	specifies whether the application is going to be operated using fingers (i.e., Pointing Granularity COARSE) or a mouse (Pointing Granularity FINE).
toolkits	specifies which graphical toolkits are supported by the device (e.g., Java Swing or HTML).
scrollWidth	specifies the maximum horizontal scroll width in multiples of the screen width (i.e., 1 means no horizontal scrolling).
scrollHeight	specifies the maximum vertical scroll height in multiples of the screen width (i.e., 1 means no vertical scrolling).

The properties specified through such an application tailored device specification are used by different modules of UCP:UI. The properties *resolution* and *dpi* together with the style information specified in the *defaultCSS* are used by the layout module presented in Section 4.3 for calculating the size of a specific widget or container. The properties *pointingGranularity* and *toolkits* specify information that can potentially be used to *filter* the transformation rules presented in Section 4.2. To filter transformation rules means here that all rules that do not satisfy a specific pre-condition (e.g., support for a specific GUI toolkit) are discarded before the transformation process is started. Our *basic* transformation rule set presented in Section 4.2 is independent of these two properties and thus ensures that each compliant Communication Model can be transformed. *scrollWidth* and *scrollHeight* are taken into account by certain optimization strategies employed for tailoring the GUI for a specific device (see Section 4.4).

Such an Application-tailored Device Specification is compliant to the CRF platform definition. It even extends this definition through the "pointing granularity" and the "scroll width" and "scroll height" concepts, which are more a kind of non-functional properties than hardware or software properties.

4.2 Transformation Rules for Device Tailoring

This section presents the conceptual design and implementation of a *basic* set of transformation rules that supports the transformation of *compliant* Communication Model (a PIM) into at

least one Structural UI Model (a PSM), as presented in [RPK13b]. A rule set that allows for the generation of different GUIs for a specific Communication Model is a pre-requisite for our automated tailoring approach, which uses an optimization search to determine an optimal GUI for a given device. We use the term platform in this section as a synonym for device, because we classify the models according to the Model Driven Architecture.

4.2.1 Towards Completeness of the Rule Set

Our first aim was to design a "minimal" set of transformation rules that supports the generation of at least one PSM (i.e., Structural UI Model) for each *compliant* PIM (i.e., Communication Model). This means that the PIM is compliant to its meta-model (e.g., specified in UML or Ecore) and satisfies all constraints that are specified in addition for this meta-model (e.g., using the Object Constraint Language (OCL)). All meta-models used in UCP:UI are expressed in Ecore. Ecore implements the MOF standard, just like UML, which means that its expressiveness is limited in the sense that only those constraints expressible according to the Ecore meta-model can be specified. For example, the type of specific attribute can be specified as integer, but it is not possible to additionally constrain the values (e.g., to 0 to 100) in Ecore (nor in UML). We used OCL and Java to express additional model constraints and to specify clearly which models are compliant and thus transformable (see [RSKF11] for further details). While it is hard to formally define completeness of such a rule set, we tried to design the rule set in such a way, that no rules are obviously lacking. Our first consideration was about the granularity of units to be transformed, which led us to identifying atomic transformation units.

4.2.1.1 Identifying Atomic Source and Target Model Units

A complete set of transformation rules allows the transformation of each compliant source model. Such a rule set is important to enable the automatic transformations for different platforms, without the extra effort of having to specify platform-specific transformations. The most obvious way towards achieving completeness of transformation rules is to provide one transformation rule that is independent of specific platform characteristics (e.g., the screen size) for each atomic source model *element*. This allows the transformation of each compliant input model, but it is typically not sufficient to achieve a high quality level of the output model. The problem is that matching only atomic source model elements does not take requirements that the output model needs to satisfy (e.g., usability) into account. Thus, it is important to define atomic model *units* with the requirements on the target model in mind. This will most likely lead to the definition of atomic source model units that consist of more than one atomic source model element.

Models are typically structured in a hierarchical way (e.g., in a tree), so that the corresponding atomic model units are somehow related. Such relations typically have an explicitly defined meaning. In high-level interaction models, such relations (e.g., Discourse Relations or Temporal Operators in task models) are used to define possible flows of interaction. However, there exists additional information in the structure of the model. For example, elements that are siblings are typically stronger interrelated than elements whose common root is not their immediate parent. It is important to transfer not only the explicit information provided by the relations, but also the additional information in the structure during the transformations, for achieving a good quality level of the target model.

Another property of interaction models that impacts the definition of atomic source model units is, who takes the initiative and starts the information exchange – is it the application or the user (via the user interface)?

Figure 4.1 illustrates the case where the System takes the initiative in a simple Question-Answer scenario. In this case, the question and the corresponding widgets that allow the user to provide the answer, are part of the same screen.

Figure 4.2 illustrates the second case, where the user takes the initiative, again in a simple Question-Answer scenario. In this case, the question is sent by the User and subsequently processed by the application to deliver the answer. This implies that the widget(s) that allow the user to send the question and the corresponding answer are displayed in two subsequent screens.

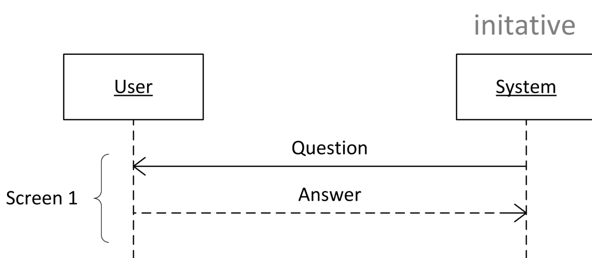


Figure 4.1: Initiative Application [RPK13b]

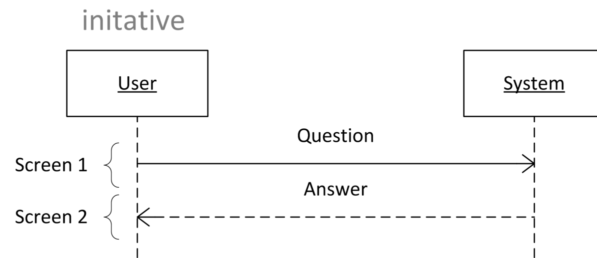


Figure 4.2: Initiative User [RPK13b]

All information that is displayed on the same screen can be transformed with the same transformation rule, so the initiative influences the size of the atomic communication units. Let us illustrate this with Discourse-based Communication Model elements. It is advisable to define Adjacency Pairs (e.g., Question-Answer) as atomic communication units in case that the System takes the initiative. This allows defining transformation rules that map the corresponding opening and closing Communicative Acts to GUI widgets and thus, to create a layout that ensures a good level of usability. In case that the User takes the initiative, a single Communicative Act is such an atomic communication unit.

The atomic source model elements in Discourse-based Communication Models are Communicative Acts and Discourse Nodes (i.e., Adjacency Pairs or Relations) that relate them. We distinguish between relations whose meta-model element has a defined number of child links (e.g., 2 for all Relations that have a Nucleus and a Satellite Link) and Relations whose meta-model element has an undefined number of links (e.g., multi-Nucleus Relations like OrderedJoint can have 1 to n Nuclei). We defined Adjacency Pairs with exactly one closing Communicative Act as atomic source model units in case that the opening Communicative Act is uttered by the System. In this case, both Communicative Acts are part of the same screen (see Figure 4.1), which allows the transformation of such Adjacency Pairs in one rule. This allows for a better layout in the transformation rule's RHS and leads to a better usability of the resulting GUI model. For Adjacency Pairs that may have 1 or 2 closing Communicative Acts, we defined Communicative Acts as atomic units. An example is an Offer Communicative Act whose closing Communicative Acts may either be an Accept Communicative Act, a Reject Communicative Act, or both. Communicative Acts are the atomic units in all cases where the User has the initiative, as the Communicative Acts belonging to such an Adjacency Pair are part of two consecutive screens (see Figure 4.2).

4.2.1.2 Taking Platform Restrictions into Account

As usual, the transformation rules enrich the models with information while concretizing them over various levels of abstraction. Interaction models define all possible flows of events, but no details regarding style or layout. Such information is added through the RHSs of the transformation rules, i.e., usability principles need to be considered in the GUI model through the corresponding characteristics (e.g., layout or style attributes). Our *basic* transformation rules define the default style for the generated GUIs and completely specify the layout for their RHSs. They can also contain hints for the layout of their RHS in its container (which is typically generated by another rule). The overall layout for each screen is created automatically, based on the information that is provided through the transformation rules (see Section 4.3 for details).

Normally, the transformation rules are *filtered* before the transformation happens, according to platform characteristics, so that exactly one rule matches each source pattern. Thus, they already contain platform-specific aspects and have to be created anew or at least adapted for a new platform. Our aim regarding the design of transformation rules for our new conceptual approach (see Figure 1.3) was to design them in a way that we can render a GUI for each compliant interaction model. This means that at least one remaining rule exists for each atomic transformation (see above). Additionally, we aim for automated tailoring through optimization search, which implies that we need to be able to generate not only one but different GUIs for the same compliant interaction model. So, our transformation rule set has to provide several rules that match the same source model pattern to enable the transformation of a specific PIM into different PSMs. To achieve both aims, the transformations must not contain platform-specific aspects (e.g., be applicable only for a certain screen size), but rather provide information that allows distinguishing them according to the optimization objectives, and there must be transformations that transform the same PIM pattern to different PSM patterns. An example are two transformation rules that match the same PIM pattern, but the corresponding PSM parts either split the resulting screen or not. This example does not relate to a specific device, but to optimization objectives that a minimum number of clicks shall be achieved.

To keep the rules independent of the platform screen size, we sort them relative to each other and not according to absolute screen size values. According to the optimization objectives in UCP:UI, relation rules need to specify whether they split the screen (e.g., using tabbed panes as container for its children) or not (e.g., using a panel as container for its children) and Communicative Act/Adjacency Pair Rules are ranked according to how much space their RHS occupies. For example, a transformation rule that creates a drop down list for the selection of a value is rated “smaller” than a list that creates a panel that displays more than one entry at the same time.

4.2.2 Basic Transformations in UCP:UI

We implemented a complete set of transformation rules that allows for the generation of more than one GUI for the same Communication Model. Which GUI Model is optimal depends on the applied optimization objectives. The currently applied objectives in UCP:UI *are maximum amount of information*, *minimum number of navigation clicks* and *minimum scrolling* [RPK+11b]. UCP:UI uses its proprietary transformation engine [PFRK12] that allows for more than one rule to match the same source pattern. This transformation engine uses its proprietary transformation language, specified in an Ecore meta-model, which we used to implement our *basic* transformation rule set. We distinguish four different rule classes:

1. Communicative Act Rules,
2. Adjacency Pair Rules,
3. Relation Rules and
4. Second Level Rules.

The first three classes were derived from our analysis of the source and target models and are so-called "first level rules". It is important to note that our Communicative Act and Adjacency Pair rules do not only match a specific Discourse- but a specific Communication Model pattern. This means that different rules match according to the propositional content of a given Communicative Act, or the Communicative Act(s) of an Adjacency Pair. The fourth class is comparable to lazy or called rules in the Atlas Transformation Language (ATL) and we call them "second level rules". Second level rules can be applied due to UCP:UI's two step transformation process [KFK09b] and allow for the definition of first level transformation rules that do not depend on a specific domain model.

Style attributes for all widgets of a transformation rule's RHS are specified in a corresponding Cascading Style Sheet (CSS). All styles referenced by *basic* transformation rules are specified in a corresponding so-called "basic CSS".

4.2.2.1 Two-step Transformations

Let us illustrate this two step process and its implications on the transformation design with a small example of a login dialog, where the System asks the User for username and password. Such a login dialog can be modeled as an OpenQuestion for the domain concept `User`, with the attributes `username` and `password`. This Adjacency Pair represents an atomic source model unit according to our definition, because the System takes the initiative.

Figure 2.11 above shows this pattern together with the corresponding GUI widgets that are (partly) generated through the transformation rule depicted in Figure 2.11. The upper part of Figure 2.11 shows the source model pattern – an Adjacency Pair with an OpenQuestion and the corresponding answer, where the propositional content of the OpenQuestion is specified as `get one EObject`, where `get` refers to the *basic* ANM and `EObject` to the Ecore meta-model, which is loaded as default DoD Model. The lower part shows the corresponding RHS. The RHS is part of the Structural UI Model and consists of the container `Panel`, a heading `Label`, a `Panel` that contains an `Input-` and an `OutputPlaceholder` and a `Button`. This rule is generic and matches all OpenQuestion-Answer Adjacency Pairs whose propositional content is of the type EObject, like the class User from our login example.

The second step iterates over all attributes of the `EObject` and calls second level rules for each attribute. In particular, the second level rules are matched to the attribute type and generates a corresponding concrete interaction widget. For our example there are two attributes: username and password, both of the type `EString`. Therefore, the OutputPlaceholder is replaced through a `Label` for each attribute name (i.e., username and password), and the `InputPlaceholder` through a `Textbox` for each attribute that allows the user to insert the corresponding values (e.g., "david", "top-secret"). The corresponding Structural UI is illustrated at the right side of Figure 2.10.

The advantage of this two step process is that it supports the specification of generic first level rules that match an `EObject` or an `EDataType`, for example. To enable the transformation of any

Ecore meta-model element that can be referenced as propositional content we created rules for `one` and `many` EObjects and EDataTypes with placeholders. These placeholder are substituted through concrete interaction widgets through calling second level rules. This means that our *transformation rules do not depend on a specific Domain-of-Discourse Model.*

We defined second level rules that create concrete interaction widgets for all data types available in Ecore (i.e., int, short, long, double, float, String, Date, Enum and boolean). These rules are defined for `Input-` and `OutputPlaceholders` and there is an additional rule that creates the `Label` for attribute names, as used in the login example. This means that we defined 19 second level rules in total, which match a different source pattern each.

4.2.2.2 Alternative Transformations

Table 4.2 shows the number of *basic* second level rules together with the number of rules that we defined for the other three categories. Table 4.2 also shows that there is variability in the Adjacency Pair and the Relation Rules. In particular, we defined two rules with the same source pattern for four Adjacency Pair patterns and 10 rules with the same source pattern for relations. These additional rules support the generation of different PSMs for the same PIM.

Table 4.2: *Basic* Transformation Rules

Category Name	Minimal Rules	Additional Rules
Communicative Act Rules	28	-
Adjacency Pair Rules	52	4
Relation Rules	23	10
Second Level Rules	19	-
	122	14

Additional Relation Rules

Relation rules typically create containers for their children. According to their behavior definition, their children are either displayed concurrently, leading to a `Panel` or in different screens, leading to a `Choice` element (e.g., a Tabbed Pane). All 10 Relation Rules that match already existing LHSs target Relations whose children can be displayed concurrently. In contrast to the minimal rules, they split the screen. Thus, the respective RHSs of the minimal and the corresponding additional rules have different impact on the resulting PSM (i.e., Structural UI Model) and are weighed differently according to the optimization objectives (a split screen means at least one additional click, but reduces the amount of scrolling).

Let us illustrate this with two rules that match the `OrderedJoint` relation. All children of an OrderedJoint can be displayed concurrently. Figure 4.3 shows the minimal rule, which in principle matches an OrderedJoint relation and creates a `Panel` containing the OrderedJoint children. Thus, the resulting GUI will show them concurrently. Figure 4.4 shows the corresponding additional rule, which also matches an OrderedJoint relation, but creates a `TabbedPane` containing its children instead. This means that less space is required in the resulting GUI model (i.e., Structural Screen Model), as the screen is split. How much space is saved in the end depends on the children and, therefore, on the specific Communication Model.

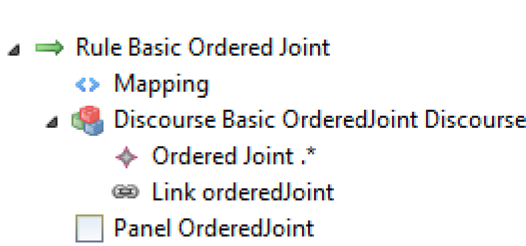


Figure 4.3: Minimal OrderedJoint Rule [RPK13b]

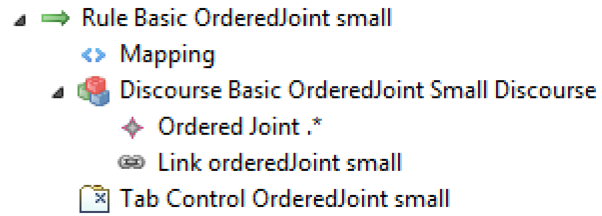


Figure 4.4: Additional OrderedJoint Rule Small [RPK13b]

Additional Adjacency Pair Rules

The LHSs of the four additional Adjacency Pair rules match the same PIM pattern as the four corresponding minimal rules, but they use different widgets in their respective RHSs. In particular, these additional rules generate drop-down lists, which show only one list entry per default instead of displaying more than one entry at the same time (e.g., using radio buttons for their selection) as done by the minimal rules.

Discussion of UCP:UI Rules

Providing more than one transformation rule for the same source model pattern and not filtering them upfront according to platform characteristics, but rather evaluating their impact on the resulting PSM, allows us to generate alternative PSMs and to determine an optimal one according to our objectives.

Evaluating the resulting GUI and not each rule separately, allowed us to design transformation rules that are independent of a specific platform. This claim can be challenged, as not all toolkits do necessarily support the widgets that our Structural UI meta-model defines, which means that a rule's RHS may also be used to filter rules. This is absolutely true for more advanced rules. In our *basic* rules, however, we only used a set of very basic widgets (e.g., labels, buttons, radio buttons) that are supported by plain HTML. As a Web-browser is available on most devices today, this means that we can render and tailor GUIs for most devices, still taking their screen real estate (i.e., size and resolution) into account.

Finally, Table 4.2 shows that 122 rules are needed to support the transformation of each compliant Communication Model. This minimal rule set contains exactly one rule for each atomic source model unit and thus, does not support the generation of alternative PSMs. The automated generation of alternative PSMs is supported through our 14 additional rules that each generate an alternative RHS for LHSs that are already matched by rules in the minimal set. Both, minimal and additional rules together constitute our *basic* transformation rule set. The number of possible PSMs depends on the number of atomic source model units found in the PIM and on how many additional rules match already existing LHSs. The number of possible PMSs has to be multiplied with the number of alternatives for each additional rule that matches. Even with only 14 rules, the number of alternatives can be large, as 10 rules match Relations that link Adjacency Pairs and are typically heavily used in Communication Models.

4.2.3 Beyond the State-of-the-Art

Transformation languages like the Atlas Transformation Language[1] (ATL) or Query / View / Transformations[2] (QVT) have been developed to support the specification of transformation rules. These languages can reference any meta-model specified in an appropriate modeling language (typically Meta Object Facility[3] (MOF) compliant languages like the Unified Modeling Language[4] (UML) or Ecore) and are thus also applicable for model-driven UI generation.

The currently available transformation engines for ATL and QVT support only matching one rule per source model pattern. Wagelaaer et al. [WVDSD10] present a solution for this problem called Module Superimposition. Their approach is based on ATL and supports overriding transformation rules. By overriding, they mean replacing the original rule with a new one, whereby it is not possible to refer to the original rule anymore. Using module superimposition in our conceptual approach, however, has the drawback that the rule set would need to be changed during the transformation process. If the rule set is modified before the transformation, there is no difference to the standard approach, because there would be exactly one rule for each source model pattern.

A GUI generation framework that supports multi-device GUI generation using ATL is UsiComp [GFCDC+12]. This framework supports transformation rule design at run- and design-time through the designer. Thus, the device information is encoded manually in the transformation rules before the transformation is performed. Another UI generation framework supports semi-automatic multi-device UI generation based on ConcurTaskTree Models (CTT) [PMM97] [PS02]. Semi-automatic means that the high-level CTT model, which is a Computation Independent Model, is tailored manually to a system-task model that already takes device characteristics like screen size into account. In a newer version of this tool [PSS09b] rendering content differently for different devices can be achieved through annotations. Other GUI generation approaches specify their transformations in Groovy[5] or Java as part of their transformation tool [PEPA08], which makes the modification of the transformations for device-tailoring or GUI beautification difficult [AVVP09].

So, multi-device GUI generation typically relies on manual creation of device-dependent transformation rules or manual device-tailoring of the source model, which makes subsequent transformations with ATL or QVT sufficient.

4.3 Automated Layout Calculation

Layout is one of the crucial aspects of GUI usability, as the following statement from the Microsoft User Experience Guidelines underlines: “Layout is the sizing, spacing, and placement of content within a window or page. Effective layout is crucial in helping users find what they are looking for quickly, as well as making the appearance visually appealing. Effective layout can make the difference between designs that users immediately understand and those that leave users feeling puzzled and overwhelmed” [Mic10]. Similar UI guidelines are provided by major software companies/platforms (e.g., Apple [App11], Eclipse[6], KDE[7], etc.), to achieve a certain level of

[1] http://www.eclipse.org/atl/
[2] http://www.omg.org/spec/QVT/1.1/
[3] http://www.omg.org/mof/
[4] http://www.uml.org/
[5] http://groovy.codehaus.org/
[6] http://wiki.eclipse.org/User_Interface_Guidelines
[7] http://techbase.kde.org/Development/Guidelines

usability for their applications and consistency between them.

The research community took up the challenge to automate layout creation in the course of model-driven WIMP-UI generation about two decades ago. For example, scientists investigated layout techniques [VG94] and defined metrics for layout appropriateness [Sea93]. However, the interest in the research community seemed to cease about a decade ago. Today most model-driven UI generation approaches use manually created presentation or layout models. Automatisms are only applied to a very limited extent. The key problems seem to be to select the right from a multitude of options and the direct impact of the layout on the usability of the generated UI. Kennard and Leany [KL10] even suggest that the layout of a widget on the screen is perhaps the most intractable issue in GUI generation, because layout "exposes a myriad of small details around UI appearance, navigation, menu placement and so on. The problem is so difficult, in fact we believe it insoluble". They argue that a non-optimal layout that compromises the final product in usability or even only aesthetics, compromises automated generation in general. We also think that the problem of automated layout creation is too complex to be solved in a generic way. It is impossible to generate a given UI without specifying the details.

Our approach aims to specify as much layout information as possible upfront, before the transformation, in a reusable way. The designer is enabled to specify layout information on the concrete UI level of the Cameleon Reference Framework [CCT+03], which we consider the most appropriate level to do so. To make these specifications reusable, they are part of the right-hand sides (RHSs) of transformation rules. These RHSs are the GUI model parts that the transformation rules create. Usually, one screen of the GUI is composed of the GUI model parts of several transformation rules. These GUI model parts are composed according to the structure of the high-level model. Such high-level models do not contain any layout specifications, thus their hierarchical structure is the only characteristic that is available for consideration for creating the layout. The rule that creates the container for the children does not consider their number, otherwise it would already be dependent on a specific high-level model.

What we propose is to enable the designer to add *hints* for each child container that specify where it shall be placed in its parent container and to complete the layout data automatically, based on these hints. Thus, we support a seamless transition from fully-automatic to semi-automatic GUI generation with human involvement. Our approach will most likely not enable the designer to generate the GUI she desires fully automatically in the first run. However, as we also think that iterative design is a crucial component of achieving high-quality user interfaces [PRS11] and aim to reduce the number of iterations that a designer needs to develop the GUI she desires in the long run.

The remainder of this section explains our automated layout approach in detail before we illustrate its integration in UCP:UI, based on [RPV12, Lei10]. We present the experiences gained through the implementation of our approach and discuss several issues that we consider worth for further investigation. We conclude this section with how our approach enhances the related state-of-the-art.

4.3.1 Our Automated Layout Approach

The problem of fitting a given amount of widgets into a given (screen) space is basically a constraint satisfaction problem. Our automated layout approach supports the specification of further constraints (i.e., in addition to the space constraint given through the Application-tailored

Device Specification) through so-called basic layout criteria and “Layout Hints”. Basic layout criteria constrain the order in which the widgets are inserted and their placement. Layout Hints can be used by the designer to further constrain the widget insertion order and to specify where to put them in their container, helping her to achieve the desired layout. We introduce a small running example of a flight selection GUI to illustrate the basic constraint satisfaction problem and our solution for achieving a desired layout in this context.

4.3.1.1 Basic Layout Criteria

WIMP-UIs are structured in the form of a tree and use containers to group widgets. The root node of such a GUI tree must be a container (e.g., a frame) and the leaf nodes are the concrete interaction widgets (e.g., buttons, labels, etc.). The intermediate levels are populated by containers that group widgets, which can again be containers or concrete interaction widgets. Each of these containers specifies the layout for its direct children. Hence, the overall screen or frame layout is composed of the layout of all its sub-containers.

Our automated layout approach exploits the hierarchical GUI structure and creates the overall screen layout bottom-up, in steps for each level. We start with calculating the size of the interaction widgets (i.e., leaf nodes), because they determine the size of their immediate container and in fact, all containers up to the root container. By creating the overall layout step-wise, we can reduce the complexity of the problem to placing a container's direct children. Creating more complex UIs is a matter of applying our technique with a larger number of steps, because more complex GUIs have a deeper hierarchical structure. This is adequate for supporting scalability.

Let us illustrate the problem of placing a container's direct children with a simple flight selection GUI, again slightly different from the examples used above, to fit our purposes in this section. The left side of Figure 4.5 depicts the widgets that are on the same hierarchical level of this GUI. In particular, they are the direct children of the container depicted at the right side of the figure. The selected level contains four buttons (*Back*, *Next*, *Home* and *Logout*) and one container (*Airport Selection*). This container comprises the two containers Select Departure Airport and Select Destination Airport, which comprise the concrete interaction widgets (e.g., drop-down lists) to select the departure and the destination airports. These widgets are not depicted and the containers are grayed out, because they are on lower levels in the GUI hierarchy.

Our layout approach is built on the assumption that the size of each widget to be placed in its container is available. We show how we satisfy this requirement in UCP:UI below. For now, the challenge is to fit the widgets displayed on the left side of Figure 4.5 into their container on the right side. Our approach inserts large things first, because smaller ones can be fit in later more easily. This is a usual heuristic in general and in constraint satisfaction problems in particular. We opted for three different criteria to determine the widget order: widget *area*, *width* and *height*. The widgets can be ordered according to one of these criteria and placed in their container, starting in the upper left corner.

After the insertion of each widget, we use a right-bottom strategy [BHLV94] to get the next placement options. Figure 4.6 illustrates this strategy with our running example. The Airport Selection container in Figure 4.5 is the widget with both, the largest width and height, and thus also the largest area. Therefore, we insert this widget first at insertion point 1 (i.e., position (0,0)). Insertion point 1 is already occupied and, therefore, removed. The new placement options for the next widgets according to the right-bottom strategy are insertion point 2 (position (361,0))

Figure 4.5: Flight Selection GUI Widgets [RPV12]

and insertion point 3 (position (0,211)). The next widget according to our criteria is one of the buttons.

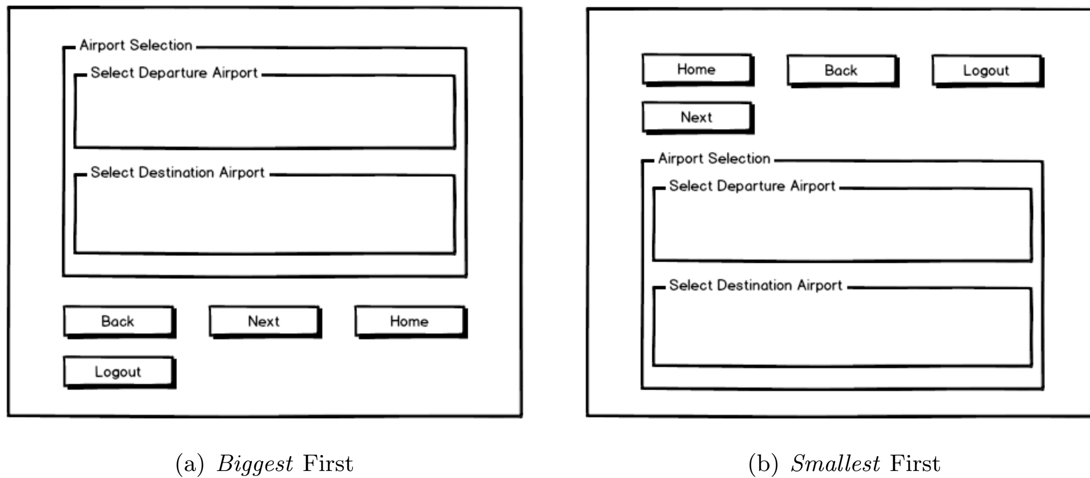
Figure 4.6: Placement Options [RPV12]

Our approach offers the designer to choose between the criteria *smallest waste space* and *best ratio*, to influence the selection of the insertion point. *Smallest waste space* means that we compare the space that is left empty in the cut containers (see the scratched out areas in Figure 4.6). *Best ratio* means that we compare the width/height ratio of the cut containers each with a given value (e.g., the display ratio of the device to render for).

This selection has be done before the layout engine is started through specifying the corresponding option in a configuration file. To compare all placement options for a new widget and to finally identify the best one for the selected criterion, we insert it at each point and subsequently cut the container at the right-most and the lowest widget edge. Next, we evaluate each insertion point according to the selected criterion.

Figure 4.6 illustrates the insertion of the back button at insertion point 3 according to the smallest waste space criterion (the second best option is crossed out). After inserting each widget, we update the available insertion points. This means that the right – insertion point 4 (position

(101,211)) – and the bottom point – insertion point 5 (position (0,241)) – are added after the insertion of the back button. The now occupied point (insertion point 3) is not available anymore and, therefore, removed. Our approach repeats these steps for each widget. This basically implements a hill-climbing search [Kai89]. In the running example, we are able to fit all widgets in the container. The resulting GUI is illustrated in Figure 4.7(a). This GUI will nevertheless not satisfy the user, because all buttons are placed beneath the Airport Selection container and the Back, Home and Logout button are typically expected to be at the top.



(a) *Biggest* First (b) *Smallest* First

Figure 4.7: Automatically Placed Widgets in Flight Selection GUI

Reversing the widget ordering criteria from biggest to smallest first creates the layout shown in Figure 4.7(b). Inserting the smallest widget first is typically not done for constraint satisfaction problems, but may still be useful in GUI generation, because command buttons are typically placed at the top of a window/screen. Unfortunately, this will also not lead to a satisfying GUI, because all buttons are now placed in a different order above the Airport Selection container and the Next button is not in the lower right part, where users may most likely expect it. These problems have two reasons. First, our criteria (i.e., smallest waste space and best ratio) cannot distinguish the buttons. This makes their order arbitrary and most likely not the desired one. Second, the next button would be placed above the Airport Selection container instead of below, where the user may expect it. Figure 4.8 shows what we consider an adequate layout for our running example.

So far, we use the ordering of the corresponding high-level model elements that they represent on the GUI as a heuristic. According to our experience, this does still not lead to the desired order in most cases. One reason may be that the order of elements does not matter for temporal operators that model concurrency in high-level models (e.g., the CTT [PMM97] concurrency operator, or the Joint in Discourse-based Communication Models [FKH+06]). The position of a widget on the screen however, is closely related to its functionality. We provide the designer with the means to provide additional information for the layout creation.

4.3.1.2 Layout Hints

To solve the problem of where to place a child widget in its container, we introduced 'Layout Hints' in addition to our basic layout criteria. Our Layout Hints can be used to assign any GUI

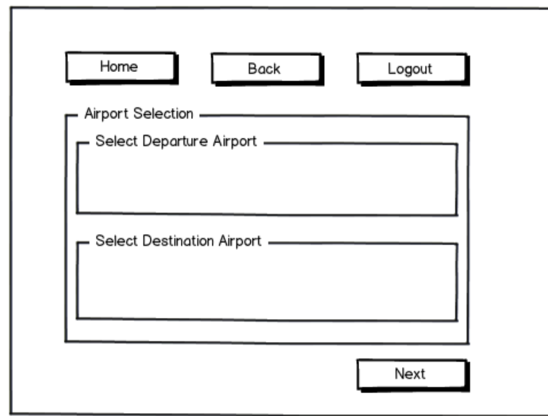


Figure 4.8: Flight Selection UI with Layout [RPV12]

widget to a certain region in its parent container. They support the divide-and-conquer principle, which means that they support partitioning a container in different regions and distributing the widgets to them. Layout Hints can be attached to GUI widgets specified in a transformation rule's RHS.

Figure 4.9 shows that we partition a container in a kind of table with three rows – TOP, MIDDLE, BOTTOM – and three columns – LEFT, CENTER, RIGHT. Thus, we can distinguish nine different regions in total. Our Layout Hints introduce the vertical alignment options *top* and *bottom* and the horizontal alignment options *left* and *right* to identify each of these nine regions. Figure 4.9 depicts how these options need to be combined to identify a certain region. The `NO_ALIGNMENT` region contains all widgets without Layout Hints. To distinguish between rows/columns and regions we use two different typesets, e.g., LEFT for rows/columns and `LEFT` for regions.

	LEFT	CENTER	RIGHT
TOP	TOP_LEFT	TOP	TOP_RIGHT
MIDDLE	LEFT	NO ALIGNMENT	RIGHT
BOTTOM	BOTTOM_LEFT	BOTTOM	BOTTOM_RIGHT

Figure 4.9: Container Partitioning [RPV12]

Our Layout Hints also support the specification of a so-called *z-index*. The z-index concept is similar to the one used in HTML/CSS and allows refining the order of widgets that need to be placed in the same region. A widget with a higher z-index gets placed first. Layout Hints can be assigned to widgets either by the designer, when creating a transformation rule, or by

the framework, when it creates new widgets during the transformation process (e.g., buttons for navigation in case that containers are split). Our layout module evaluates these Layout Hints and places each widget in the specified region of its container.

The regions of a container are rarely equally populated. A common scenario is a crowded `NO ALIGMNENT` region and empty `LEFT` or `RIGHT` regions. This means that the width and height of each region have to be adapted for each container. Our approach uses two steps to perform this adaptation. First, we distribute the container's height between TOP, MIDDLE and BOTTOM rows. The width of each row is equal to the container's width. In the second step, we distribute each row's width between their LEFT, CENTER and RIGHT column. We use the minimum area required by each column/region (i.e., the sum of all its child widget areas) for this calculation. The minimum area for a container is simply the sum of all its child areas and can always be calculated, as the size of the leaf nodes is already available after a transformation rule has been matched. We then use our basic layout approach presented above to place the widgets in each column/region row-wise, from `TOP_LEFT` to `BOTTOM_RIGHT`. We additionally set the horizontal alignment option (i.e., left, center or right) of the widgets according to the column (i.e., LEFT, CENTER or RIGHT) of their region.

After each column/region, we refine the width calculation for the subsequent column(s)/region(s). After we finish one row, we refine the height calculation for the subsequent row(s). In this way we create the layout for each container stepwise, through placing the corresponding widgets in each of its regions. Let us illustrate this approach with our running example. Table 4.3 shows the Layout Hints that we attach to the four buttons to achieve the desired (adequate) layout shown in Figure 4.8. The *Home Button* is assigned to the `TOP_LEFT` region, the *Back Button* to the `TOP` region, the *Logout Button* to the `TOP_RIGHT` region and the *Next Button* to the `BOTTOM_RIGHT` region. There are no Layout Hints for the *Airport Selection Container*, which is, therefore, placed in the `NO_ALIGNMENT` region.

Table 4.3: Layout Hints for Adequate Flight Selection GUI

Widget	**Layout Hint**
Home Button	alignment-y: top alignment-x: left
Back Button	alignment-y: top
Logout Button	alignment-y: top alignment-x: right
Next Button	alignment-y: bottom alignment-x: right

Our new approach starts with distributing the available screen height (specified in the given Application-tailored Device Specification) between the TOP, MIDDLE and BOTTOM rows (according to the ratio between the corresponding widget areas). Next the TOP row is partitioned, again based on the ration between the corresponding widget areas. In fact, the TOP ROW is equally partitioned in the `TOP_LEFT`, the `TOP` and the `TOP_RIGHT` regions, because the corresponding buttons have the same area. After the *Home Button* has been placed in the `TOP_LEFT` region and its horizontal alignment has been set to "left", there are no more widgets to be placed in this region. So, our approach subtracts the actually occupied width from the screen width and re-distributes the remaining width between the `TOP` and the `TOP_RIGHT` regions (again equally, because the buttons have the same area). This is repeated after placing the *Back Button*.

All widgets in the TOP row have been placed after the *Logout Button*. So, our approach subtracts the actually occupied height from the screen height and re-distributes the remaining height between the MIDDLE and the BOTTOM rows, using the ration between the corresponding widget areas. Next we partition the screen width between the `LEFT`, the `NO_ALIGNMENT` and the `RIGHT` regions. In fact, there are no widgets to be placed in the `LEFT` and the `RIGHT` regions, so the width of the `NO_ALIGNMENT` is equal to the screen width. Next, we place the *Airport Selection Container* in this region.

All widgets in the MIDDLE row have been placed after the *Airport Selection Container*. So, our approach again subtracts the actually occupied height from the screen height and assigns the remaining height to the BOTTOM row. Next we partition the screen width between the `BOTTOM_LEFT`, the `BOTTOM` and the `BOTTOM_RIGHT` regions. In fact, there are no widgets to be placed in the `BOTTOM_LEFT` and the `BOTTOM` regions, so the width of the `BOTTOM_RIGHT` is equal to the screen width. Next, we place the *Next Button*, including the setting of its horizontal alignment to “right”, and calculate the width and height of the container based on the now available layout.

In our example, each region contains only one widget, which definitely creates the desired layout. If more than one widget is assigned to a specific region (e.g., using the alignment-y *top* for the Home, Back and Logout buttons), we could still achieve the desired layout through different z-index values.

Our example illustrates that our Layout Hints enable the designer to capture non-functional requirements concerning the layout that cannot be captured in high-level models.

Layout Hints supposedly demand less effort, in comparison to a manually created layout model (e.g., in form of a GUI mock-up), if they are reusable. This is the case, as they are incorporated in the RHSs of transformation rules.

Without any Layout Hints, the problem is exactly the same as our basic layout problem presented above. This means that the use of Layout Hints potentially allows the designer to improve the resulting GUI (in the sense that less manual customization is required to achieve the desired one), in comparison to the layout created through our basic approach.

4.3.2 Automating Layout in UCP:UI

We integrated our new layout approach in UCP:UI in two steps. First, we extended the transformation language meta-model to incorporate our Layout Hints. Second, we extended the framework's layout module [Lei10]. In particular, we refined its widget size calculation and adapted it to incorporate our Layout Hints.

Widget Size Calculation

We use a depth-first algorithm to traverse the GUI structure and create the layout bottom up. This means that only the size for non-container widgets has to be provided, as they are the leaf nodes of the UI tree. The size of containers is calculated after their layout has been created.

The most direct way to specify width and height of a widget is to set the corresponding attributes of the widget in the transformation rule. Additionally, we support the specification of width and height through Cascading Style Sheets (CSS). CSS support style specifications via id, class and element. However, designers rarely use this option, because most transformation rules transform

content that is only known at the time when they are applied and not at the time when they are designed. This means that a size specification at rule design time is not possible. To add flexibility in this regard, we introduced support for only width or height specifications. We then calculate the missing value automatically during the transformation process. This calculation is either based on static text, or default character numbers for widgets that handle text. Static text can be specified in the transformation rule, or is completed at the time of transformation. Default character numbers can be specified at the corresponding attributes in the domain model elements.

Width and height of a container are calculated after its widgets were placed. List widgets are special containers in this regard. Their size, except for drop down lists, is determined by how many of their entries shall be visible. Our approach creates the layout for one entry and calculates the list's size, using default values for the number of visible entries. The layout module also uses default values for each widget type's width and height in case that no other specification is available. All default values are defined in a configuration file, which can be adapted by the designer according to her needs.

Creating the Layout in UCP:UI

The integration of our layout approach in UCP:UI is illustrated in Figure 4.10. UCP:UI supports the automated transformation of compliant *Discourse-based Communication Models* to *Structural Screen Models*, using the *Basic Transformation Rules* presented in Section 4.2. The layout module is part of the transformation framework, among other modules, and provides the implementation of the concepts for automated layout calculation presented above. The designer can provide *More specific Transformation Rules* with Layout Hints and configure the layout module to achieve a desired GUI (i.e., Structural Screen Model).

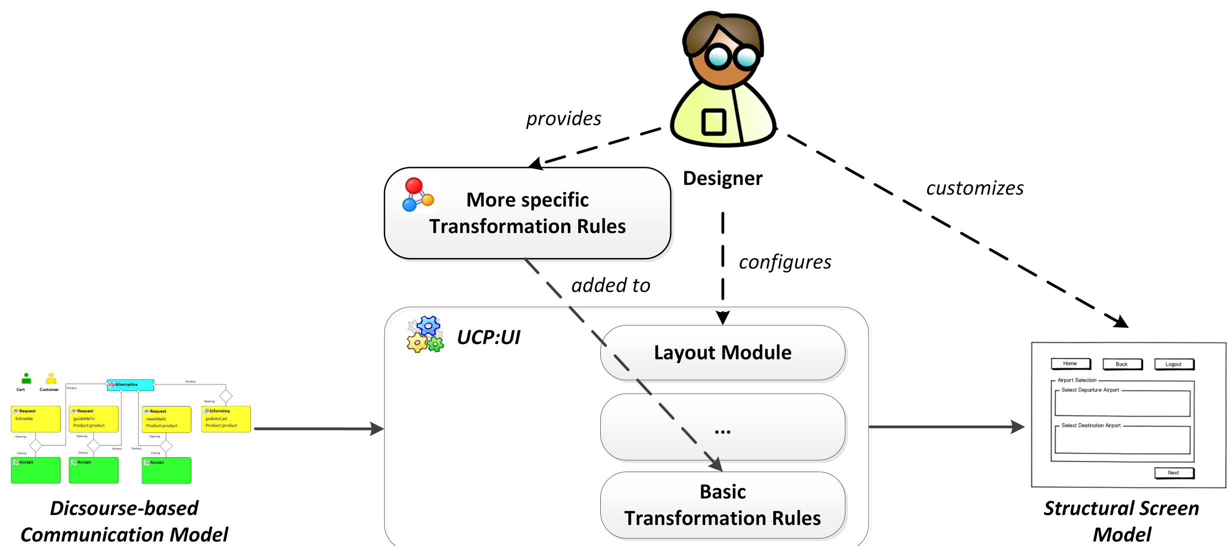


Figure 4.10: Integration of our Approach in UCP:UI, based on [RPV12]

Table 4.4 summarizes the configuration options for the designer. The *scrolling* options do not contain the value *none*. The reason is that this would lead to containers that have no valid layout, in case that the widgets do not fit in the given size. The available scrolling options ensures that

our approach is able to create a layout for any container. This option is used by our automated tailoring strategies, presented in Section 4.4. The *widget ordering* options influence the sequence in which widgets are inserted in their container. The designer can use the z-index in our Layout Hints to directly influence this widget order. The *insertion strategy* options are used to sort potential insertion points for a widget. Which insertion points are available is influenced by the scrolling option. Finally, the designer can adapt the *default widget property* values for width and height of each widget type.

Table 4.4: Layout Module Configuration

Property	**Options**
scrolling	both, horizontal, vertical
widget ordering	biggest/smallest first according to width, height, area
insertion strategy	smallest waste space, best ratio
default widget properties	width, height

GUIs are frequently developed for a specific device with a given display size. This means that information like the size of the root container (i.e., frame or screen), which typically matches the given display size, is usually available, while the size of its sub-containers is typically unknown. Designers rarely use the option to explicitly specify the size for a sub-container at design time, because the exact number or type of its child widgets depends on the structure of the high-level model and is not available at rule creation time. Our Layout Hints resolve this problem, because they are attached to the child element instead of the containers. However, an exact size calculation of a container is impossible without a valid layout. So, we need the exact size of each container to calculate its layout and we need the layout to calculate its exact size. This is the well-known chicken-and-egg problem. We address this problem by using the *best ratio* option as fall-back, in case that no size is available for a container. We obtain the size of the root container from our Application-tailored Device Specification and calculate the size for each region based on the minimum area of the direct children as described above.

Figure 4.10 also shows that UCP:UI provides a set of *basic* transformation rules (already presented in Section 4.2). GUIs that are generated with the *basic* rule set are fully functional and allow early testing of the application logic, but they hardly ever satisfy the user.

So, Figure 4.10 shows that the designer can additionally provide more specific transformation rules. These transformation rules are added to the existing rules and usually render application-domain specific concepts in the desired way. If the designer used mockups or wireframes during an initial design phase, she can include the layout specifications in the GUI model part (RHS) of a specific transformation rule. Direct layout specifications can be defined only within one GUI model part (i.e., RHS) of a transformation rule. This means that direct specification is not possible in case that one transformation rule creates the container, and other transformation rules create the UI model parts that populate it. Our Layout Hints can be used in this situation, to roughly specify the placement for a child in its container.

Let us illustrate the effect of our Layout Hints with our running example. There are additional implications through the two-step transformation process of our transformation engine on the

layout creation, which we will also illustrate using this example. Figure 4.11 shows a *basic* transformation rule for transforming a *ClosedQuestion-Answer Adjacency Pair*. The upper part of the figure shows the Communication Model pattern that this rule matches (i.e., the rule's left hand side (LHS)). The root of the LHS is the Discourse element, which contains an Adjacency Pair with an opening Communicative Act of the type *ClosedQuestion* and the corresponding *Answer* as the closing Communicative Act. The content of the Closed Question is specified as `many EObject`. This discourse pattern models the interaction, where the system provides a list of objects and the user selects one. Our running example uses this construct to model the airport selection. The objects are Airport concepts with the two String attributes *name* and *code*.

The GUI model part created by this transformation rule (i.e., its RHS) is depicted in the lower part of Figure 4.11. The root container of this rule is the `ClosedQuestion-Answer` panel. The attached Layout Hint assigns this panel to the top region of its parent container. Its direct children are the `Heading` label, the `ClosedQuestion` foldout list and the `Send Answer` button. A fold out list is a container, which in case of our transformation rule contains an abstract `Output Placeholder`. All Placeholders are substituted through concrete widgets during the second step of the transformation process [KFK09b]. Placeholders are quite useful, because they support the specification of general transformation rules, independently of specific domain elements. On the other hand, they make the direct layout specification within one UI model part of a transformation rule tricky, because it can be replaced by more than one widget.

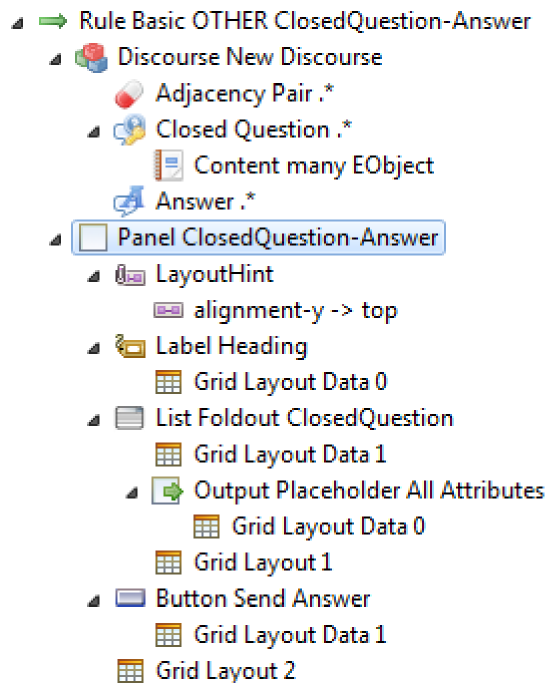


Figure 4.11: ClosedQuestion-Answer Transformation Rule [RPV12]

Figure 4.11 shows that the layout has been specified explicitly for each widget in the rule's UI model part. All containers use a `Grid Layout` and their direct children have the corresponding `Grid Layout Data` attached. Each placeholder contains a so-called "content‘" property (not visible in the graphical rule representation in Figure 4.11), which specifies an OCL statement that is evaluated on the domain model object transformed by this transformation rule during the second transformation step. This step creates concrete interaction widgets according to the

result of the OCL statement. For example, the OCL statement *"eAllAttributes"* leads to the creation of one widget for each attribute of the domain model object that is transformed. In case of our running example a label for the (Airport) *name* and another one for the (Airport) *code* are created. At rule creation time, however, layout data can only be specified for the placeholder widget. In case that a placeholder is replaced by more than one widget, the layout needs to be completed and adapted after the second transformation step.

In UCP:UI the transformation is completed before the resulting GUI is passed on to the layout module. It is not possible for the layout module to distinguish between widgets that have been created in the first or the second step. This means that partial layout is difficult to achieve without adding extra information during the transformation. We opted for the solution to adjust the layout data directly after each second transformation step. To achieve this, we added the possibility to specify a *replacement direction* (either *vertical* or *horizontal*) for each placeholder widget. After the second step has been completed, we add layout data to the widgets created by this step and adjust the layout data for all widgets that need to be moved due to the new ones. We additionally added the possibility for the designer to explicitly specify whether the widgets within a specific container shall be placed (i.e., laid out) or not after the transformation process has been completed, through a setting a property on a specific container in the transformation rule's RHS. This ensures that the manually specified layout in the transformation rule is preserved. Furthermore, we gain performance, because our approach can easily recognize whether a layout shall be created or if the size of the container can be calculated directly.

The gray part of Figure 4.12 shows the resulting GUI after the application of our transformation rule. Our Grid Layout Data offers similar characteristics as Java Swings's GridLayoutData. This means that alignment and fill options can be defined in addition to the positioning (i.e., row, column, rowspan, colspan). The drop-down list shows that the Output Placeholder was substituted by two labels, and that the layout was adjusted horizontally (as defined by the corresponding attribute of the Output Placeholder).

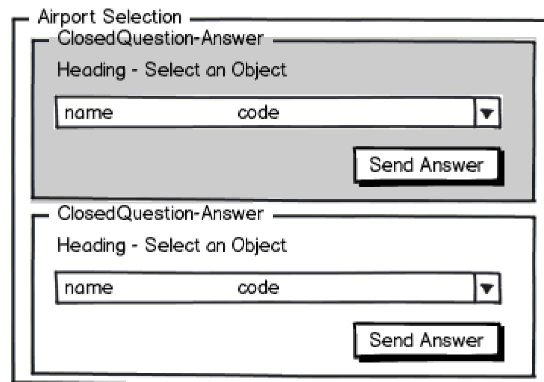


Figure 4.12: Transformation Result [RPV12]

If we apply the same rule again to transform another Adjacency Pair (i.e., the one that models the `ClosedQuestion` for the Destination Airport) we would create the lower part of Figure 4.12. The repeated application of the same transformation rule assigns the destination airport selection again to the top region of the container. If the designer wanted to explicitly assign it to the bottom, she would have to copy the rule and adjust the Layout Hints of the container.

Another problem where our Layout Hints are quite useful, are the two Send Answer buttons. Here, the user would expect a single button to send both answers (i.e., selected airports) at the

same time. Our framework combines these buttons based on the dynamic model of the GUI when the Screen Model is generated (as described in Section 3.2). In effect, it creates a new button that sends all answers. This button is added to the original buttons' closest common parent for which a layout is created. Our Layout Hints provide the possibility to assign this button to the bottom right region, which is presumably the area where a user would expect a button to submit her data or to proceed to the next screen.

Figure 4.8 sketches the concrete UI for our running example. This GUI results from applying the transformation rule in Figure 4.11 twice, combining the *Send Answer* buttons in Figure 4.12 to a *Next* button and using the Layout Hints of Table 4.3 in the transformation rules that create the *Back*, *Home* and *Logout* buttons. The Layout Hint for the *Next* button is automatically attached by the framework when it combines the *Send Answer* buttons (i.e., when generating the Screen Model).

Our running example illustrates that the initial layout is brought to GUI models in UCP:UI in three steps. First, layout data can be defined manually in the UI model part of each transformation rule. Second, the layout data is completed and adjusted by the transformation framework after the completion of each second transformation step. Third, missing layout data is calculated and added by our layout module. Figure 4.10 shows that the designer can finally perform additional layout modifications on the GUI model (i.e, Screen Model). This is important, as we do not think that a specific layout can be generated fully automatically (i.e., without additional input through the designer) and we, therefore, want to provide a good entry point to compensate still existing shortcomings.

The remainder of this subsection discusses two issues regarding our widget ordering options that we consider worth addressing, and underlines the decisions that we took during the development of our approach. These options determine the order in which the widgets are inserted in their respective container and, therefore, has a major impact on the resulting layout.

We conjecture that it is not feasible to find ordering options, which lead to a satisfying layout in general, because the placement of widgets is highly dependent on their functionality and, therefore, application-specific. With our Layout Hints, we opted to support the manual specification of additional layout information, which allows the designer to influence this order and to achieve the desired layout. Alternatively, this information could be derived automatically from a specific widget's functionality and previous work showed that widget classes can be derived systematically from different task types [KJ02]. However, our problem is different, as we need to distinguish different widgets of the same class (e.g., different buttons) according to their functionality. Additionally, there is currently no standard available to specify a widget's functionality to the best of our knowledge and automatically deriving the position from such a specification would still be based on layout heuristics and most likely require additional manual customization anyway.

Another idea in this context is to reduce the impact of the ordering strategy through a second iteration that *automatically evaluates and improves* the outcome of the first layout cycle based on existing evaluation criteria [BHLV94] and techniques [LFN04]. This would even add more flexibility as different criteria or techniques could be applied. Many evaluation approaches [PBM+11, FBK+08], however, need to be applied at run-time, as they are based on log data which is not available at design time. Additionally, they focus more on the interaction technique than on the layout. The most important problem with automated evaluation for us is that the feedback depends on the evaluation technique and criteria [BHLV94]. Therefore, a powerful approach has to provide different placement strategies and a variety of evaluation options. We put this idea out of scope for this doctoral dissertation, because in general, it will take some

time for the designer to figure out the right criteria and technique and in the worst case, this will take even longer than the manual customizations would have taken (assuming that the resulting layout is the same).

Human intervention requires adequate support. We claim that manipulating layout data in our graphical tree editor is more illustrative than specifying grammar-based layout constraints, but we agree that a graphical "what you see is what you get" editor, probably similar to the Gummy [MVLC08] GUI builder, would bring a considerable improvement. Such an editor should support the creation of transformation rules and the manual adaptations of the resulting GUI model through direct manipulation of layout and style data.

4.3.3 Beyond the State-of-the-Art

The Layout of a GUI can be created at design- or at run-time. Compared to each other, design-time approaches better support human intervention and run-time approaches require less effort. This subsection contrasts design- and run-time approaches that paved the way for our own work.

Lok and Feiner present a survey of automated layout techniques for information presentation in [LF01]. They state that "A presentation's layout can have a significant impact on how well it communicates information to and obtains information from those who interact with it." Furthermore, they underline the necessity to consider high-level relationships and spatial constraints, and to provide a grammar-based language to specify them. The authors admit that, however powerful and expressive grammars may be, they may be difficult to use.

Vanderdonckt and Gillo summarize techniques from the area of visual design, to make them exploitable for GUI design [VG94]. Their work gives a good overview of the multitude of options and provides layout guidelines to cope with the corresponding complexity. Further suggestions for a dynamic strategy for computer-aided visual design are provided in [BHLV94]. This work compares different placement strategies and concludes that in any case, generation may be considered as a fair starting point for manual refinement.

Designer intervention is supported by DON, the user interface presentation design assistant [KF90]. This approach applies design rules to generate menu and dialog box presentations, which can be customized by the designer to influence widget positioning.

Most model-driven GUI development approaches or frameworks that operate at design time use manually created layout or presentation models [PEPA08, PSS09b]. Noteworthy in this context is SketchiXML [CFK+04], which provides an approach to transform hand-drawn UI sketches, also known as wireframes or mock-ups, in GUI specifications based on UsiXML[8]. This approach does not work on the GUI layout, but enables designers to express such a layout without much effort.

A model-based approach for layout generation at run-time is introduced by Feuerstack et al. [FBSA08]. Their layout model consists of a list of ordered statements, which is interpreted at run-time. A layout model generator with a graphical editor is provided to facilitate the layout model creation for the designer. The run-time interpretation of the layout model makes their approach adaptable to new contexts of use, but it also makes the resulting GUI less predictable for the designer. This is also true for the approach of Keränen and Plomp [KP02], who adapt existing layouts to devices with a smaller screen size through splitting at run-time.

[8] `http://usixml.org`

The Automated Interface Layout (AIL) [LF02] approach creates the layout of digital library query results for a given screen size. This approach works at run-time and assigns vertical space to each query result, before creating its horizontal layout. AIL automatically scrolls vertically if insufficient screen real estate is available. SUPPLE, by Gajos et al. [GWW08], is another run-time GUI generation approach that automatically generates the GUI layout based on a hierarchical model of widgets, having weights for selection and placement. Gajos et al. suggest that users may be willing to sacrifice predictability if the alternative has a large-enough improvement in adaptation accuracy.

Kennard and Leaney [KL10] define five key characteristics needed by any UI generation technique before it should expect wide adoption, and built their Metawidget approach upon them. Of particular interest in the context of layout is their fifth principle: "applying multiple, and mixtures of layout". They explicitly negate the use of heuristics, as heuristics would decrease the usability of the resulting UI. Furthermore, heuristics would make the approach more unpredictable for the designer, which is especially problematic if you try to replace manually created UIs with identical, but generated ones (i.e., retrofitting). For these reasons, they exclude automated layout generation from their focus and consciously limit their approach to deriving only layout facts that are constrained by the back-end architecture. This is possible, as their Metawidget approach operates at run-time.

Run-time approaches in general have the advantage that more information is available (e.g., number of list entries). Rendering and layout decisions can be based on this knowledge. The drawback is that the outcome is less predictable for the designer. Moreover, layout creation at run-time may lead to a performance degradation. Our approach is applied at design-time, to give the designer control over the *look & feel* of the resulting GUI, without having to create an application-specific layout model.

In general, the layout of a GUI is a crucial component for good usability. Fully-automatic layout creation is possible. The downside is that such UIs will hardly ever be satisfying for the end user. The problem is that it is hard, if not impossible, to generate a specific GUI based on heuristics. Specifying details however, is a cumbersome and labor-intensive task. Our approach uses heuristics that can be selected by the designer through configuration. Moreover, we provide the option to specify additional information in the form of Layout Hints, which are part of a transformation rule's GUI model part. This means that they are reusable through reapplication of the transformation rule. The GUI model part of such a transformation rule is on concrete UI level, which we consider the most appropriate one for layout specifications. The integration of our approach in UCP:UI demonstrated the feasibility of our concepts and showed that it is sensible to accomplish the layout stepwise. Our approach can be used to efficiently create GUIs for rapid prototyping and initial user evaluation. It is independent of a specific high-level model and can thus be incorporated in any model-driven UI generation framework using transformation rules.

4.4 Automated Device Tailoring

GUIs need to take device characteristics like available screen space into account to achieve a good level of usability. A GUI can be tailored for a specific device through optimizing it according to given optimization objectives and device-specific constraints. This requires that alternative GUIs have to be evaluated and in this way (at least in principle) compared with each other. To achieve

this, the generation approach needs to provide more than one GUI for the same Discourse-based Communication Model.

This GUI optimization/tailoring problem is comparable to the classical *knapsack problem*. So, let us briefly summarize the knapsack problem, according to [MT90]:

> "Suppose a hitch-hiker has to fill up his knapsack by selecting from among various possible objects those which will give him maximum comfort. This *knapsack problem* can be mathematically formulated by numbering the objects from 1 to n and introducing a vector of binary variables $x_j (j = 1, \ldots, n)$ having the following meaning:
>
> $$x_j = \begin{cases} 1 & \text{if object } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$
>
> Then if the *profit* p_j is a measure of the comfort given by object j, w_j its size and c the size of the knapsack, our problem will be to select, from among all binary vectors x satisfying the *constraint*
>
> $$\sum_{j=1}^{n} w_j x_j \leq c,$$
>
> the one which *maximizes* the *objective function*
>
> $$\sum_{j=1}^{n} p_j x_j.$$
>
> "

In our case of automated GUI generation, we need to fit a given amount of widgets that belong to the same screen into the screen space of the given device. Hence, this screen corresponds to the *knapsack*, constrained through the available screen space, and the widgets are the *objects* that we need to fit in. A widget has a defined size (corresponding to the *weight* of an object), and its *profit* depends on its utility according to given optimization objectives. The utility of a widget is hard to define and we conjecture that it depends on its context. Widgets that require additional clicks (e.g., tabbed panes) result in a higher GUI cost, according to our optimization objectives [RPK+11b]. So, we determine a *cost* value for a specific GUI based on the contained widgets. This cost value is dual to profit (i.e., $cost = -profit$).

4.4.1 Search Space

Our *basic* transformation rule set, as presented above, consists of *minimal* rules, which support the transformation of any compliant Communication Models, and *additional* rules, which match the same source model patterns as minimal rules but create different target patterns. The *basic* rule set and UCP's transformation engine allow for more than one rule to be matched [PFRK12]. This achieves the generation of more than one GUI for a given Communication Model, where each GUI is defined through some combination of instantiated transformation rules, and in fact, defines a search space.

Let us illustrate such a search space through a small Communication Model excerpt from a vacation planning application and our *basic* transformation rules. Figure 4.13 shows the Discourse Model for the initial interaction of this application, which we use as a running example through-out this section. In fact, it is a Communication Model excerpt, as there are references to DoD Model and ANM concepts in the propositional content (not visible in Figure 4.13) of the Communicative Acts. However, we only use the Discourse Model in our running example, because it is sufficient to illustrate our tailoring approach. In particular, this model presents a welcome message ("Willkommen" in German), modeled through the Informing Communicative Act, and allows for the selection of an article or an event from two corresponding lists ("Artikel" or "Veranstaltungen", respectively, in German), modeled through the two ClosedQuestion-Answer Adjacency Pairs (ADJs).

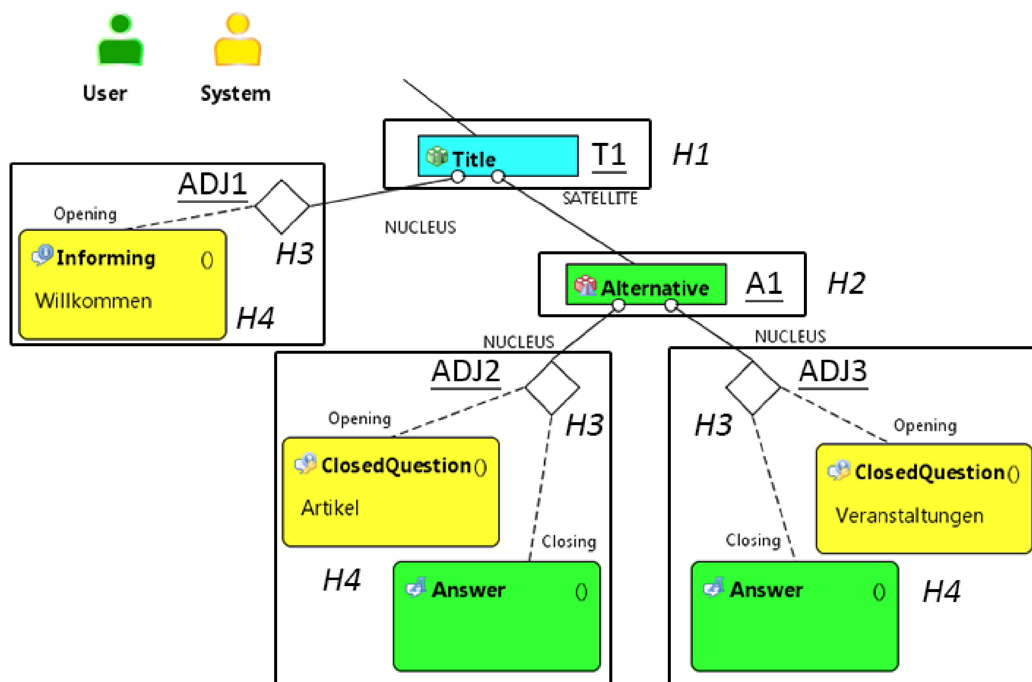


Figure 4.13: Vacation Planning Initial Interaction (Start Screen) Discourse Model Excerpt

The rectangles in Figure 4.13 mark the five Communication Model patterns that are matched by *basic* transformation rules. Table 4.5 lists them with their patternIDs in the first column and the matching *basic* transformation rules with their ruleIDs in the second column. It shows that more than one transformation rule is matched for patterns A1, ADJ2 and ADJ3 each. Note that the *basic* rules shown in the second column of Table 4.5 are more precisely instantiations of transformation rules, which result from matching a *basic* rule to the Communication Model pattern defined in its LHS.

Figure 4.14 illustrates the search space defined by the Communication Model excerpt in Figure 4.13 and our *basic* rule set. This search space has a tree structure. The tree's leaf nodes represent GUIs that can potentially be generated, while the other nodes on the paths represent only partly constructed GUIs. The edges are labeled with "patternID, ruleID". For example, the search tree includes only one edge for T1 and ADJ1, respectively, because each of these patterns is only matched by a single *basic* transformation rule. A1 is matched by two *basic* rules (R2

Table 4.5: Transformation Rules for Vacation Planning Start Screen Discourse Model

Pattern Identifier	*Basic* Rule(s)
T1	R1
ADJ1	R4
A1	R2, R3
ADJ2	R5, R6
ADJ3	R5, R6

and R3), resulting in two branches. ADJ2 and ADJ3 are also matched by two *basic* rules each, resulting in eight GUIs (GUI 1 . . . GUI 8) that can be generated for our example.

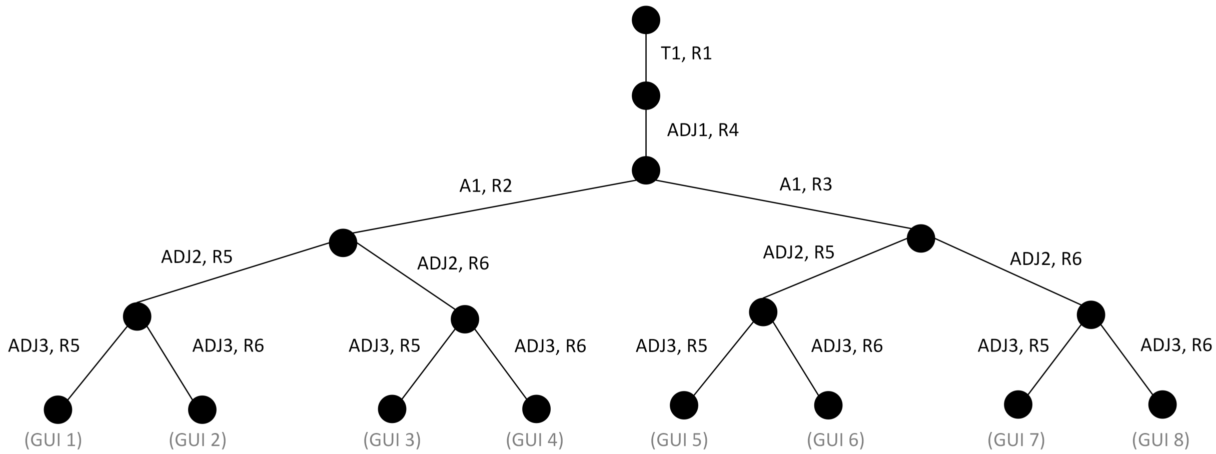


Figure 4.14: Search Tree for Vacation Planning Start Screen Discourse Model

In general, we can calculate the number of GUIs that can be generated for a given Communication Model as,

$$GUInumber = \prod_{i=1}^{n} rulenumber_i$$

where n is the number of source patterns that are matched in the Communication Model and $rulenumber_i$ is the number of transformation rule instantiations that match this pattern. Using our *basic* set of transformation rules, this number is larger than 1 for all those source patterns matched through *additional* rules.

4.4.2 Objective Function

An objective function is a mathematical formulation of a given optimization problem. It is used to assign a numerical value to any given solution of the problem, according to the optimization objectives.

We formulated our optimization/GUI tailoring problem through three optimization objectives, which are [RPK+11b]:

1. maximum use of the available space,

2. minimum number of navigation clicks, and
3. minimum scrolling (except list widgets).

Our objective function calculates a *cost* value that reflects a given GUI's achievement of our objectives, so we use the term cost function instead of objective function below. An optimal GUI according to our cost function is one with lowest cost according to our objectives 1 and 2 that satisfies our constraint, given through the screen size (i.e., available screen space) specified in the Application-tailored Device Specification of the device to tailor for. In fact, this is a soft-constraint because of objective 3. Allowing for scrolling enables us to generate a GUI for any compliant Communication Model, even if no GUI fits the screen.

Checking this constraint requires the Structural Screen Model to be available. The generation of the Structural Screen Model requires calculating the complete layout, which is computationally intensive. To reduce this computational effort, we designed our cost function in a way that allows calculating a cost value of a given GUI through evaluating the transformation rules that are used for generating it. This allows comparing all GUIs that can potentially be generated before actually generating these GUIs and calculating the complete layout.

Evaluating the transformation rules that are executed to generate a specific GUI means that our cost value cannot reflect the space finally required by the GUI. This cost value needs to reflect the optimization objectives so that we can sort the transformation rule combinations for all GUIs that can be generated in theory *relative* to each other, without needing to determine the GUIs' actual sizes.

To consider objective 1, we need to estimate the demand of screen space of the transformation rule's right hand side (RHS). However, estimating the finally required amount of screen space is not feasible at rule design-time, because the Communication Model is not available yet. So, we compare the RHSs of transformation rules with the same LHS, and sort them *relatively* to each other. To estimate which rule has the higher cost according to objective 1, we need to estimate which RHS requires less screen space than the other ones that match the same LHS, because we want to minimize the overall cost while maximizing the use of the available space.

We defined a *relativeSpace* property for each transformation rule, which allows us to sort all transformation rules with the same LHS. This *relativeSpace* property is specified through an `integer` value and has been set to a specific value c for all *minimal basic* transformation rules, because their LHSs are mutually exclusive. When a new transformation rule is added, this property has to be set either to a higher or lower value, depending on whether the rule's RHS requires more space or less than the corresponding transformation rule from the *minimal basic* rule set. An example is the *additional basic* transformation rule that renders a single-selection item list as a drop-down box (instead of a radio button list), where both rules match `Closed-Question Answer Adjacency Pairs` with the content `EObject`. We assigned this *additional basic* rule a *relativeSpace* value of $c - 1$, because its RHS requires less screen space compared to the RHS of the corresponding *minimal basic* rule.

To consider objective 2, we need to evaluate whether the transformation rule's RHS requires additional navigation clicks or not. This is defined as a `boolean` value of each transformation rule through a rule property named *splitting*. `True` means that the RHS splits the screen (e.g., through generating a tabbed pane) and `false` signifies that the screen is not split (e.g., through generating a panel), which requires fewer navigation clicks than the split version.

To consider objective 3, we need to determine the exact size of the resulting GUI. However, it cannot be calculated through evaluating the transformation rules, because it requires the complete layout to be available. So, this objective can only be evaluated after generating the GUI and completing the layout. However, it is indirectly considered through considering objective 1 and 2, because a smaller space value signifies less space consumption and more navigation clicks also result in less space consumption through splitting the screen. Both mean potentially less scrolling. The importance of objective 3 lies in making our constraint, given through the available screen space, a soft-constraint and allowing us to generate a tailored GUI for each compliant Communication Model.

Considering only the *relativeSpace* and the *splitting* property for calculating the rule cost potentially results in a lot of rule combinations (GUIs) with identical cost, although their finally required space differs and they apply splitting for different Communication Model elements. To distinguish such GUIs, we additionally consider the type of the matched Communication Model elements (i.e., the rule's LHS) and the hierarchical position of the matched Communication Model pattern's root node in the Discourse Model tree when calculating the cost for a specific rule combination. Both are available before the GUI actually needs to be generated.

So, a specific GUI ($\overrightarrow{r}$) is characterized thought a specific combination of k transformation rule instantiations r_i, which are executed to generate it. We denote such a GUI as a vector $\overrightarrow{r}$:

$$\overrightarrow{r} = \begin{pmatrix} r_1 \\ \vdots \\ r_k \end{pmatrix}$$

We defined the $GUIcost$ of a specific combination of transformation rule instantiations ($\overrightarrow{r}$) as:

$$GUIcost(\overrightarrow{r}) = \sum_{i=1}^{k} rulecost(r_i)$$

We defined the cost of a specific transformation rule instantiation r that matches a Communication Model pattern as:

$$rulecost(r) = -w_{relspace} * relspace(r) + w_{splitting} * splitting(r) + w_{level} * level(r)$$

The first addend $w_{relspace} * relspace(r)$ represents objective 1 and has a negative prefix, because this objective is to maximize the use of the available space, while the overall objective is to minimize the cost. $relspace(r)$ is an `integer` value that depends on the *relativeSpace* property of the transformation rule r. So, the value of $relspace(r)$ reflects the relative ordering of the transformation rules established through this property. $relspace(r)$ is 0 for all *minimal basic* transformation rules where *relativeSpace* is c and not 0 otherwise.

The second addend $w_{splitting} * splitting(r)$ represents objective 2 and has a positive prefix, because this objective is to minimize the number of navigation clicks. $splitting(r)$ is an integer value that depends on the type of Discourse element that is split (i.e., matched by the rule's LHS). In particular, there are 8 Discourse Relations (i.e., `Background`, `OrderedJoint`, `Joint`, `Alternative`, `Elaboration`, `IfUntil`, `Switch`, `Condition`) that potentially allow for splitting

and we use this mechanism to reflect the semantic meaning of a specific relation through different weights. Assigning different $splitting(r)$ values for splitting `Background` or `Elaboration`, for example, can be used to define that splitting `Background` has a lower cost than splitting `Elaboration` relations. Our rationale here is that the interaction specified in the `Background Satellite` does not necessarily require interaction to proceed in the discourse and will always be displayed. An `Elaboration Satellite`, in contrast, is only displayed if a specific condition is fulfilled and is typically not available initially. If this `Satellite` is displayed in an additional tab after a screen change, it will be occluded by the still available tab for the `Nucleus` and might, therefore, not be noticed by the user. We defined splitting procedural constructs (i.e., `IfUntil`, `Switch` and `Condition`) with higher costs than the other relations, because the relation between their child-nodes is typically less obvious to the user and is, therefore, more difficult to grasp if not presented on the same screen. $splitting(r)$ is 0 for all *minimal basic* transformation rules where *splitting* is `false` and not 0 otherwise.

The third addend $w_{level} * level(r)$ is used to distinguish rule combinations where at least one transformation rule has been matched that is not part of the *minimal basic* rule set. In particular, $level(r)$ reflects the hierarchy level of the matched pattern's root node in the Discourse Model tree. For example, splitting a `Joint` relation on a lower level has a higher cost than on a higher level. Our rationale for this choice is that elements that are related through their immediate parent are more closely related than elements via more remote ancestors (e.g., the root node) and splitting them should, therefore, have a higher cost. This addend is 0 for all *minimal basic* transformation rules (i.e., *relativeSpace* is c and *splitting* is `false`) and not 0 otherwise.

So, a GUI that is generated through *minimal basic* transformation rules has the cost 0 by definition. This value is user defined and does not reflect the finally required space by the corresponding GUI. The cost value is only used for ordering the GUIs relative to each other.

Each addend includes an additional weight (i.e., $w_{relspace}$, $w_{splitting}$, w_{level}), which can be used to influence the impact of a specific addend on the overall *rulecost*. They can be used to assign different weights to distinguish the objectives (i.e., to strictly minimize the use of available space before splitting a screen), or to allow for a trade-off between objectives 1 and 2.

Let us illustrate our cost function with the Discourse Model excerpt shown in Figure 4.13 and the corresponding instantiations of *basic* transformation rules shown in Table 4.5. These rules are explicitly separated into instantiations of the *minimal basic* and the *additional basic* transformation rules in Table 4.6. The given pattern identifiers in addition to the rule identifiers underline that these are transformation rule instantiations. The last column of this table additionally shows the hierarchy level for each transformation rule, as assigned by our currently implemented cost function.

Table 4.6: Transformation Rule Instantiations for Vacation Planning Start Screen Discourse Model with Hierarchy Levels

Pattern Identifier	*Minimal Basic* Rules	*Additional Basic* Rules	Hierarchy Level
T1	R1_T1	-	1
ADJ1	R4_ADJ1	-	3
A1	R2_A1	R3_A1	2
ADJ2	R5_ADJ2	R6_ADJ2	3
ADJ3	R5_ADJ3	R6_ADJ3	3

Our cost function assigns the levels *bottom-up*, so that all Communicative Acts are on the same hierarchy level and that the root node is always on level 1 (i.e., the lowest level) by definition. Subsequently, the highest costs are assigned to this level and the lowest costs to the level of the root note. Our rationale for this choice was, again, that elements that are related through their immediate parent are more closely related than elements via more remote ancestors (e.g., the root node) and splitting them should, therefore, have a higher cost. Let us illustrate the assignments of hierarchy levels through our running example Discourse Model depicted in Figure 4.13, assuming that the `Title` relation is the root node of our Discourse Model tree. This Discourse has 4 hierarchy levels in total. Assigning the levels bottom-up and giving them the highest level puts all `Communicative Acts` on hierarchy level 4, denoted as *H4* in Figure 4.13. The corresponding `Adjacency Pairs` are assigned to level 3 (*H3*), the `Alternative` relation to level 2 (*H2*) and the `Title` relation as root node to level 1 (*H1*).

Alternatively, we could assign the levels *top-down*, which results in the same number of levels, but potentially different population through the Discourse Model elements. For example, the Communicative Acts and Adjacency Pairs are typically assigned to different levels. Let us illustrate the different level population through our running example. Assigning the levels *top-down* results, again, in 4 levels and puts the `Title` relation on level 1. Level 2, however, is now populated by `Alternative` relation and the `Adjacency Pair ADJ1`. Level 3 is populated by the `Informing` Communicative Act and the `Adjacency Pairs ADJ2` and `ADJ3`. Level 4, finally, contains the `ClosedQuestion-Answer Communicative Acts` of `ADJ2` and `ADJ3`.

Our rationale for assigning the levels *bottom-up*, and thus putting all Communicative Acts on the same level, is that the hierarchy level should not impact the costs of rendering alternatives (i.e., non *minimal basic* transformation rules) for concrete interaction widgets.

The search space for our running example contains 8 alternative GUIs (see Figure 4.14) defined by the corresponding transformation rule combinations. Our cost function assigns a cost value to each of these rule combinations, allowing for ordering them. So, after applying our cost function we have the following ordered list of these 8 GUIs: GUI 1, GUI 5, ..., GUI 8. This ordering reflects our optimization objectives and can be calculated entirely based on the transformation rules (i.e., before generating the corresponding GUIs).

GUI 1 is the first GUI in this ordered list and is defined through *minimal basic* transformation rules (i.e., R1, R4 and R5). The corresponding rule instantiations for GUI 1 are:

$$\overrightarrow{GUI\ 1} = \begin{pmatrix} R1_T1 \\ R4_ADJ1 \\ R2_A1 \\ R5_ADJ2 \\ R5_ADJ2 \end{pmatrix}$$

All such rules specify the *relativeSpace* property with value *c* and *splitting* as false. This means that the *rulecost* for each such rule instantiation is 0 and, thus, also the corresponding *GUIcost*.

The current implementation of our cost function does not separate objective 1 and 2 strictly, but allows for a trade-off between them with a preference to rather split a screen than to strongly reduce the use of available space. So, GUI 5 is the next GUI in our list, because it matches the *additional basic* transformation rule R3 on `Alternative A1`, which splits the screen, instead of the *minimal basic* transformation rule R2. The corresponding rule instantiations for GUI 5 are:

$$\overrightarrow{GUI\ 5} = \begin{pmatrix} R1_T1 \\ R4_ADJ1 \\ R3_A1 \\ R5_ADJ2 \\ R5_ADJ2 \end{pmatrix}$$

So, its cost is calculated as:

$$GUIcost(\overrightarrow{GUI\ 5}) = rulecost(R3_A1) = w_{splitting} * splitting(R3_A1) + w_{level} * level(R3_A1)$$

This cost is completely determined through the cost of R3_A1, because all other matched rules are still *minimal basic* transformation rules with *rulecost* 0.

The GUI with the highest cost, i.e., the last GUI in our ordered list, is GUI 8. The corresponding rule instantiations for GUI 8 are:

$$\overrightarrow{GUI\ 8} = \begin{pmatrix} R1_T1 \\ R4_ADJ1 \\ R3_A1 \\ R6_ADJ2 \\ R6_ADJ3 \end{pmatrix}$$

Its *GUIcost* is calculated as:

$$GUIcost(\overrightarrow{GUI\ 8}) = rulecost(R3_A1) + rulecost(R6_ADJ2) + rulecost(R6_ADJ3)$$

Our current cost function uses weights that allow for a trade-off between objective 1 and 2. The factor $splitting(r)$ in the splitting addend, however, is calculated dynamically based on the number of elements of a certain relation, to ensure that a given "splitting order" is achieved. This "splitting order" is determined through a so-called "splitting-weight", which can be assigned by the designer to each Discourse Relation that can potentially be split. For example, if splitting a `Background` relation has a lower weight than splitting a `Joint`, this addend ensures that splitting all `Background` relations in a given Communication Model has a lower cost in total than splitting a single `Joint` relation. Assigning the same splitting-weights for different relations makes them indistinguishable for our cost function.

In general, it is possible that the same cost value is calculated for more than one rule combination. Each combination that has the lowest cost is optimal according to our cost function, so there may be more than one optimal solution.

4.4.3 Heuristic Constraint Optimization Search

We implemented a heuristic constraint optimization search approach along the lines of branch-and-bound search [LD10] to identify an optimal GUI according to our cost function. In particular, we use the available screen space given in the Application-tailored Device Specification as cut-off condition to identify potential rule combinations early that violate our constraint, represented as soft-constraint through objective 3. This is important as using a branch-and-bound approach potentially allows reducing the computational effort required.

In comparison to the classical knapsack problem, our problem is additionally restricted in two ways. First, not all widgets potentially provided in a specific GUI toolkit are available for a specific screen, but only those required for the corresponding interaction defined in the Communication Model. Second, the widgets for a specific screen cannot be combined in all possible ways, because their layout is already partly specified in the right-hand side (RHS) of the transformation rule that creates these widgets.

The first restriction allows considering the transformation rules as "objects" instead of the widgets, and the second restriction allows calculating the *minimum area* required by the corresponding GUI part even before it is generated and the layout is calculated. We calculate this *minimum area* required by the RHS of a specific transformation rule through summing up the areas of all widgets (e.g., labels, buttons or text fields). The size of these widgets is already completed through our layout module when the rule is matched (see Section 4.3 above). The *minimum area* is a lower-bound, because the *actually required area* after a specific container has been laid out will always require this area or more. Our lower-bound can be formulated as:

$$minimumArea \leq actuallyRequiredArea$$

We want to avoid scrolling, so our constraint is given through the available screen space specified in the Application-tailored Device Specification. In particular, the *availableArea* is calculated through multiplying the values for `x-resolution` and `y-resolution`. So, our cut-off condition can be formulated as:

$$minimumArea \leq availableArea$$

In principle, it does not matter which search strategy (e.g., depth-first or a breadth-first) is used to traverse the search space. However, traversing it bottom-up (i.e., starting with the transformation rules that match Communicative Acts or Adjacency Pairs) allows identifying cut-offs early, because the *minimumArea* for each node of our search space tree can be calculate immediately. Starting at the root node, in contrast, requires the evaluation of all nodes in a specific branch until the first one is reached where concrete interaction widgets are specified in the corresponding transformation rule's RHS.

Let us illustrate the advantage of traversing the our search space bottom-up with the Vacation Planning Discourse Model excerpt shown in Figure 4.13 above and the matched *basic* rules shown in Table 4.6 above. Our *basic* transformation rule set provides two rules (R5 and R6) that match the `Adjacency Pairs` `ADJ2` and `ADJ3`. R5 creates a radio button list for the selection of a specific object (i.e., an `Article` for `ADJ2` and an `Event` for `ADJ3`) and R6 creates a drop-down list, which requires less space. `Alternative A1` in Figure 4.13 is matched through two rules (R2 and R3). R2 generates a panel for its sub-branches and R3 generates a tabbed-pane (i.e., the splitting property

of R3 is true). This means that for R2, the size of its two child branches is summed-up, whereas for R3 only the area of the largest child branch is used. So, depending on the split property of a specific transformation rule, either the largest area of a specific child branch (i.e., splitting is true), or the sum of all child areas (i.e., splitting is false) is used as *minimumArea*. The resulting search space contains 8 transformation rule combinations and is shown in Figure 4.14 above. Traversing this search space bottom-up means that the transformation rules for the Discourse Model leaf nodes (i.e., `Communicative Acts` and `Adjacency Pairs`) are matched first. Such rules generate widgets required for the corresponding interaction (e.g., labels, buttons, etc.). The size for this widgets is available, which means that the *minimum area* can be calculated immediately.

Let us assume that we want to generate a GUI for a device with a *small display* (e.g., a smartphone) and that R5 is matched first on `ADJ3`. Let us further assume that the *minimum area* calculated for R5's RHS violates our cut-off condition. So, R5 can be discarded immediately for `ADJ3` and 4 search tree branches are cut off (i.e., GUI 1, GUI 3, GUI 5 and GUI 7). Next, R6 is matched on `ADJ3` and the cut-off condition evaluated again. We assume here that there are no more violations, so the finally resulting search space contains four branches (i.e., GUI 2, GUI 4, GUI 6 and GUI 8).

Alternatively, we want to generate a GUI for a device with a *large display* (e.g., desktop PC). Again, R5 is matched on `ADJ3` first, but this time the cut-off condition is not violated. Assuming that in this case only the area sum for matching R5 to `ADJ2` and `ADJ3` violates our cut-off condition, only GUI 1 in the left most branch (labeled ADJ3,R5) is discarded. So, for the large device, the finally used search space contains seven rule combinations (GUI 2 . . . GUI 8).

Traversing the search space top-down means that transformation rule R1 on `Title T1` is matched first. This transformation rule specifies a container in its RHS for which the size cannot be estimated, because it is not defined yet which concrete interaction widgets (e.g., labels, buttons or text fields) will be contained. Next, R4 is matched on `ADJ1`, which we assume here does not violate our cut-off condition. Next R2 and R3 are matched on `Alternative A1`, both specifying containers in their respective RHSs. Again, the cut-off condition cannot be evaluated, because the it is not defined yet which concrete interaction widgets will be contained. The cut-off condition can again be evaluated for the next nodes, where R5 and R6 are matched for `ADJ2`. From our example above, we already know that the RHSs of R5 and R6 together with the RHS of R4 does not violate our constraint, because GUI 2, GUI 4, GUI 6 and GUI 8 are contained in both resulting search spaces. So, the cut-offs, for both devices, will only be detected when R5 and R6 are matched on `ADJ3`. At this point in time, the complete search space has already been created. The finally resulting search spaces are of course the same, but the computational effort involved is higher, because more nodes had to be analyzed to identify cut-offs.

Finally, we apply our cost function to all remaining rule combinations and sort them according to their costs. So, the result of our heuristic search is an ordered list of transformation rule combinations that each correspond to one branch of the corresponding search space tree. This order is a *relative* order of the corresponding GUIs based on the space estimates in the rules *relativeSpace* and *splitting* properties and our cost function.

4.4.4 Automated Device Tailoring Process

Our automated process for device tailoring uses our heuristic constraint optimization search approach and is based on the Screen Model and the cost function introduced above. We implemented this process in UCP:UI.

Figure 4.15 shows this process with a focus on the tailoring loop, which consists of 4 steps (depicted as yellow rounded rectangles labeled 1 to 4). It does not depict the generation of the WIMP-UI Behavior Model, as this step has to be completed only once, before the automated tailoring is started.

Step *"Create Search Space of Rule Combinations and their Evaluation"* (0), depicted in the left side of Figure 4.15, contains the creation of the search space and the evaluation of all relevant rule combinations with the cost function. With "relevant" rule combinations we mean all combinations that were not cut off through our branch-and-bound mechanism while creating the search space. This step has to be completed only once and is a prerequisite for the automated tailoring loop. It is, therefore, not part of the loop. This step results in an ordered list of GUIs. These can be generated through executing the corresponding transformation rule combination and calculating the layout. We use here the search space for the device with the larger display, which specifies 7 GUIs that can potentially be generated (GUI 1 was already cut off). So, GUI 5 is the rule combination with the lowest cost and GUI 8 is the one with the highest cost in our list.

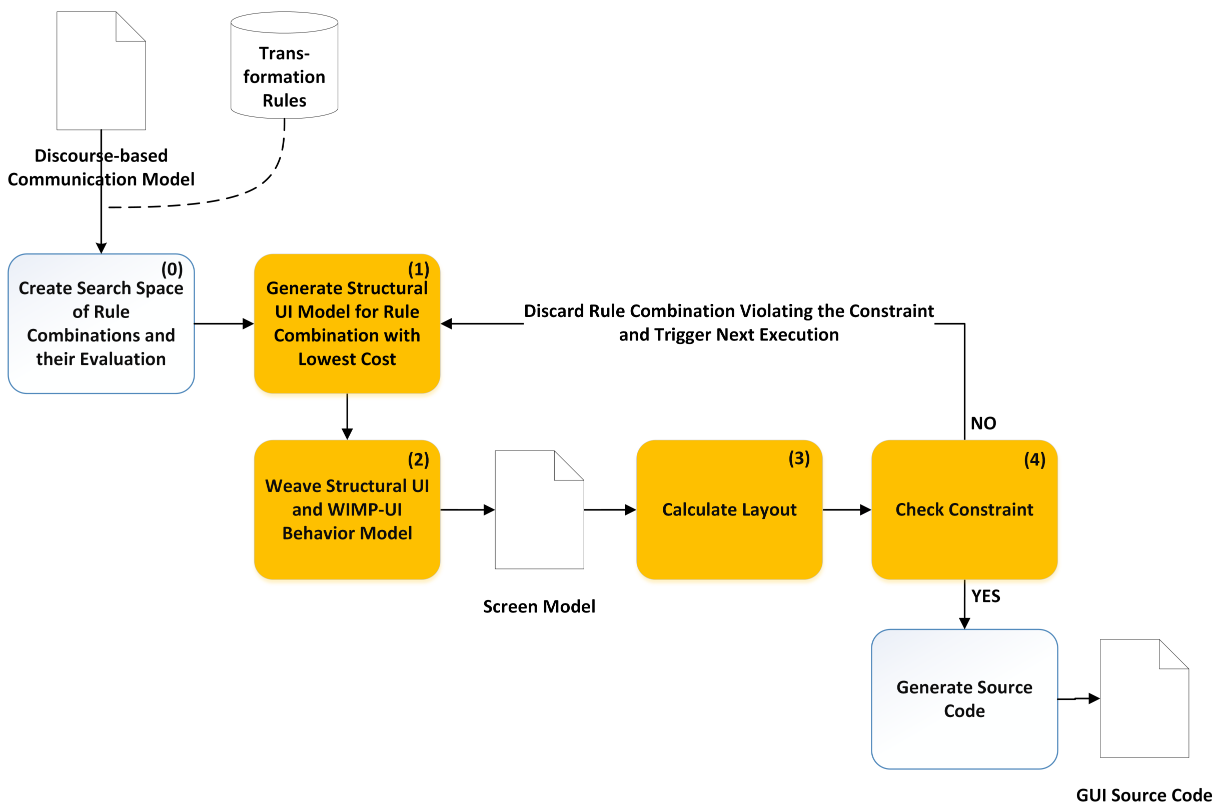


Figure 4.15: Automated Tailoring Process

Our automated tailoring loop starts with step *"Execute Rules for Rule Combination with Lowest Cost"* (1), which generates the GUI for the transformation rule combination with the lowest cost. In particular, it creates the corresponding Structural UI Model, which is weaved with the WIMP-UI Behavior Model in step *"Weave Structural UI and WIMP-UI Behavior Model"* (2), resulting in the Screen Model. Step *"Calculate Layout"* (3) calculates the layout for each screen, including the calculation of all container sizes. The execution of these three steps in each loop is required as it is not possible to determine whether a certain rule combination really violates our constraint without knowing the exact size of each screen, which is provided by the Screen Model.

Step *“Check Constraint”* (4) checks whether the size of each screen fits the space constraint of a given device, as specified in the corresponding Application-tailored Device Specification.

In our running example, we generate GUI 5 in this first loop. To keep the example simple, we selected a Discourse Model excerpt that corresponds to one Presentation Unit (i.e., all Communicative Acts are concurrently available). This means that the Screen Model that is generated with *minimal basic* rules (i.e., the transformation rule combination with the lowest cost) displays all widgets concurrently and contains only one screen. This would have been the discarded GUI 1. Our tailoring process potentially splits such Presentation Units into several screens, which actually happens in GUI 5 through the use of the *basic alternative* transformation rule R3 for the `Alternative A1`. So, the Screen Model for our running example has two screens, which correspond to the same Presentation Unit. Our constraint is checked for each of these screens. Let us assume that both screens of GUI 5 satisfy our constraint. In this case we already found an optimal solution according to our cost function and the source code generation is triggered, following the path labeled “YES” in Figure 4.15.

In case that at least one of the screens does not fit, we take the next rule combination from our list, discard the current combination and trigger the *“Execute Rules for Optimal Rule Combination”* step (1) again (following the path labeled “NO” in Figure 4.15). If there is no next rule combination, the current combination is an optimal one as it still fulfills objective 3 and the source code generation will be triggered, following the path labeled “YES”. In our running example, if all combinations violate our constraint, we will generate and check all 7 GUIs of our ordered list and finally deliver GUI 8 as result. The GUI with the highest cost is still optimal according to objective 3, because it requires the least screen space. The reason is that all potentially reducible space has been reduced and all potentially splittable screens have been split in this GUI.

In fact, the benefit of our screen-based tailoring approach can only be gained if an application has more than one screen, which process-oriented applications (e.g., booking applications) typically have. So, let us extend our running example with a second screen to illustrate this benefit.

Figure 4.16 shows a Discourse Model excerpt for payment. In particular, the `System` asks the `User` for a billing address and credit card details, modeled through the `OpenQuestion-Answer Adjacency Pairs` and presents a summary of the booking details, modeled through the `Informing` Communicative Act. All this information is concurrently available, as specified through the `Background` and the `OrderedJoint` relations, and may be on the same screen. This screen could be related with the Discourse Model for the Start Screen (shown in Figure 4.13) through a `Sequence` relation, for example.

The Payment Discourse Model consists of five Communication Model patterns that are matched by *basic* transformation rules. Table 4.7 shows the pattern identifier in column one, the matching *minimal basic* transformation rule instantiation in column two, and potentially available *basic alternative* transformation rules in column three.

If no rule combinations can be cut off, the complete search space with $2^5 = 32$ different combinations (i.e., GUIs that can be generated) has to be considered. However, let us assume again that one rule combination for the start screen could be discarded through our branch-and-bound mechanism. So, our ordered list contains 31 transformation rule combinations sorted according to their costs. The rule combination with the lowest cost is the one that contains only *minimal basic* transformation rules, except for `Alternative A1`:

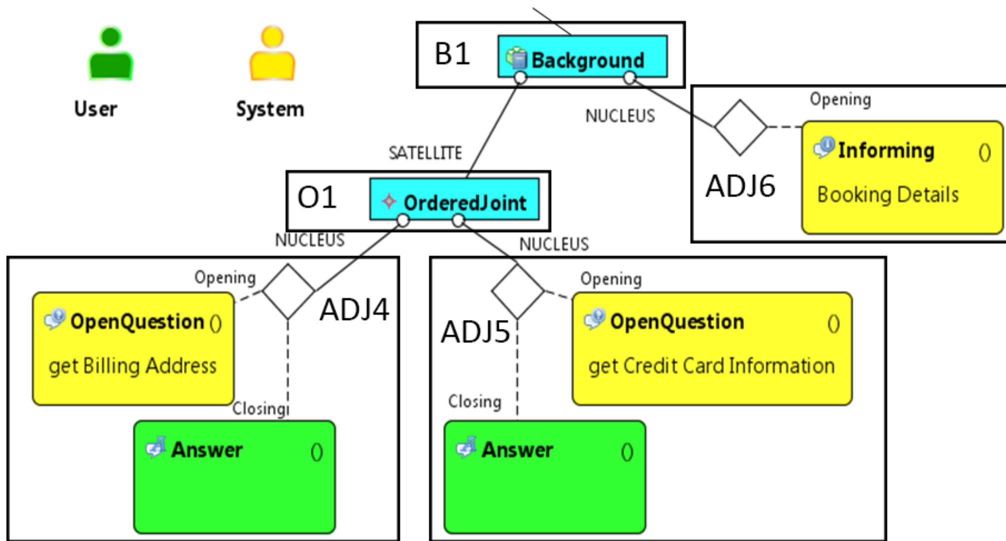


Figure 4.16: Vacation Planning Payment Discourse Model Excerpt

Table 4.7: Transformation Rules for Vacation Planning Payment Discourse Excerpt

Pattern Identifier	*Minimal Basic* Rules	*Additional Basic* Rules
B1	R6_B1	R7_B1
O1	R8_O1	R9_O1
ADJ4	R10_ADJ4	-
ADJ5	R10_ADJ5	-
ADJ6	R4_ADJ6	-

$$\overrightarrow{GUI\ 0} = \begin{pmatrix} R1_T1 \\ R4_ADJ1 \\ R3_A1 \\ R5_ADJ2 \\ R5_ADJ2 \\ R6_B1 \\ R8_O1 \\ R10_ADJ4 \\ R10_ADJ5 \\ R4_ADJ6 \end{pmatrix}$$

So, GUI 0 is generated first through executing the steps 1 to 3 in our tailoring process. In step 4 the constraint is checked for each screen. First the two screens for the Start Screen are checked, which we assume here, again, satisfy the constraint. So, an optimal solution according to our cost function has already been found for these screens and we can discard all solutions for these screens with higher cost. This means that for our running example the list is reduced to 1/8 (i.e., 8 rule combinations), because 8 different GUIs can theoretically be generated for the Start Screen Discourse Model. We assume here that the Payment Screen violates our constraint. Discarding all other combinations for the Start Screen reduces the number of rule combinations with a higher cost in our ordered list to 3, because the current combination can already be discarded too.

So, another optimization loop (following the path labeled "NO" in Figure 4.15) using the next transformation rule combination is triggered.

In this second loop, we check each screen again. However, the implementation of our approach skips already fitting screens to reduce the computational effort. So, we do not check the Start Screen again, but only the Payment Screens, which have been split and which we assume now to fit. As there are no more screens that violate our constraint, we found an optimal solution according to our cost function and trigger the source code generation (following the path labeled "YES" in Figure 4.15).

To further reduce the computational effort, our approach also skips screens that do not fit and for which no alternative rule combinations are available.

If all screens of a given Screen Model violate our constraint, the rule combination with the highest cost will be delivered as resulting GUI, because still a minimum amount of scrolling is achieved through reducing the screen space usage and introducing additional navigation clicks.

In fact, *screen-based* device tailoring treats each GUI screen as a separate knapsack problem. Treating each screen separately improves the computational performance of our tailoring approach, as all other solutions for a specific screen can be discarded as soon as the first solution that satisfies our constraint has been identified. In fact we aim to find an optimal solution according to the cost function given above for each screen, while keeping the number of screens to a minimum. This is reflected this through objective 2 (i.e., minimum number of navigation clicks), which means a minimum number of screens. Such a problem can be compared to a combination of the knapsack and the bin-packing problem [MT90].

4.4.5 Screen-based Device Tailoring Strategies

We developed four strategies that provide different options for automatically tailoring a GUI for a specific device. All of them implement our heuristic constraint optimization approach and consider a different combination of Application-tailored Device Specification attributes each. The first strategy does not allow for scrolling, which is compliant with desktop UI guidelines [Nie93, Bev05, Mic10, WMW+12]. The second and the third strategy allow for scrolling in either vertical or horizontal direction, which is again compliant with UI guidelines, but this time for touch-based devices [App12, Mic12]. We achieved the compliance of these two strategies to our objectives through a simple trick presented below. The fourth strategy allows for scrolling in both directions and configures the layout so that it aims for keeping the ratio given through the Application-tailored Device Specification. We added this strategy for the sake of "completeness" but never applied it so far, because two-dimensional scrolling is not convenient for the user. It may, however, be interesting to investigate for touch-based devices that allow zooming in and out quickly.

4.4.5.1 Screen-based Device Tailoring without Scrolling

This tailoring strategy does not allow for scrolling. In particular, it tries not to exceed the width and height of the device screen, specified through the properties `x-resolution` and `y-resolution` in the Application-tailored Device Specification of the application to render for. The layout module is configured accordingly to avoid scrolling, if possible.

4.4.5.2 Screen-based Device Tailoring with Vertical Scrolling

The "Screen-based Device Tailoring with *Vertical* Scrolling" strategy allows for vertical scrolling up to a multiple of the screen height, specified through the property `scrollHeight` in the Application-tailored Device Specification. So, the screen size is assumed to be `x-resolution` and `y-resolution`$*$`scrollHeight`, using the values specified in the Application-tailored Device Specification. Accordingly, the available screen area for our branch-and-bound approach (i.e.,*availableArea*) is calculated through multiplying the values for `x-resolution`, `y-resolution` and `scrollHeight`. This simple trick allows us to use the same implementation of our tailoring approach as for the "Screen-based Device Tailoring without Scrolling" strategy. The layout module is, again, configured to potentially allow for vertical scrolling.

4.4.5.3 Screen-based Device Tailoring with Horizontal Scrolling

The "Screen-based Device Tailoring with *Horizontal* Scrolling" strategy allows for horizontal scrolling up to a multiple of the screen width, specified through the property `scrollWidth` in the Application-tailored Device Specification. So, the screen size is assumed to be `x-resolution` $*$ `scrollWidth` and `y-resolution`, using the values specified in the Application-tailored Device Specification. Accordingly, the available screen area for our branch-and-bound approach (i.e.,*availableArea*) is calculated through multiplying the values for `x-resolution`, `y-resolution` and `scrollWidth`. Again, this simple trick allows us to use the same implementation of our tailoring approach as for the "Screen-based Device Tailoring without Scrolling" Strategy. The layout module is again configured to potentially allow for horizontal scrolling.

4.4.5.4 Screen-based Device Tailoring with Two-dimensional Scrolling

The "Screen-based Device Tailoring with *Scrolling*" strategy allows for vertical and horizontal scrolling up to a multiple of the screen height/width, specified through the properties `scrollHeight` and `scrollWidth` in the Application-tailored Device Specification. The layout module is again configured to avoid scrolling, but given the screen height as product of `y-resolution` and `scrollHeight` and the screen width as product of `x-resolution` and `scrollWidth`. In addition it is configured to keep the ratio specified through these x and y values, to facilitate zooming in and out. The layout module is again configured to potentially allow for two-dimensional scrolling.

4.4.6 Beyond the State-of-the-Art

The previous version of UCP already provided an automated GUI tailoring approach [RPK+11b], which evaluated the GUI as a whole without considering different screens. Our new *screen-based* device-tailoring is better than this approach in several regards.

Our current implementation provides four different tailoring strategies, which facilitate the exploration of alternatives for different devices, in contrast to only one previously available strategy.

Screen-based device tailoring is better in terms of computation effort than our previous approach. The major reason is that our current approach analyzes the results of the constraint check, which entails two advantages compared to our previous approach. First, if a screen already fits, we keep

it and discard all other rule combinations that apply different rules to this screen, as we already found an optimal solution. This reduces the number of remaining rule combinations if more than one combination is applicable for the given screen. If not we can skip checking the screen in future automated tailoring iterations. Second, the latter also applies for non-fitting options, if there is no alternative rule combination for the non-fitting screen we can skip checking it in subsequent automated tailoring loops.

For our previous approach neither the Screen Model nor an explicit GUI behavior model were available. This made an exact layout calculation infeasible, as the combination of panels of a specific screen was only available at run-time. Based on the Screen Model, our current implementation is able to calculate the size of a screen exactly apart from the exact sizes of lists. List sizes are calculated based on an estimation of the entry numbers, because the exact number is only available at run-time.

Furthermore, a screen-based strategy guarantees an optimal rule combination for each screen, whereas our previous approach was not able to consider which screen was violated, because the Screen Model was not available. In fact, our previous approach delivered the GUI with the highest cost if there was one screen that violated the constraint and for which no alternatives were available. Our screen-based approach, in contrast, still delivers an optimal solution for the remaining screens, according to our cost function, if they do not violate the constraint.

SUPPLE [GW04, GWW08, GWW10] is another approach that treats GUI generation as an optimization problem. In particular, it supports the generation of optimal GUIs for specific user abilities or devices, based on functional GUI models. These specify what functionality should be exposed to the user. How this functionality is to be rendered is determined through different types of constraints. Compared to interaction models, such functional GUI specifications are on a lower level of abstraction and provide less flexibility for generating concrete UIs [GWW10].

SUPPLE operates at run-time and automatically adapts the GUI through parameterizing the cost function based on so-called user traces or explicit user feedback collected through a tool named ARNAULD [GWW10]. This tool lets the user perform several tasks with different GUI renderings initially and configures the cost function automatically, based on the user's performance. Such an automated parametrization mitigates the problem that parametrizing such a cost function is tedious and error prone work [GW05], because the effect of a parameter change is hard to anticipate. Nevertheless, it is still hard to achieve a specific customization, because the effect of a specific user trace or performance on the GUI is still hard to anticipate.

Our approach, in contrast, operates at design-time, which allows the designer to keep control over the resulting GUI while performing the customizations using UCP:UI.

SUPPLE supports the adaptation of direct manipulation GUIs (e.g., a text-processor GUI) for users that have not been considered as target users by the original GUI designers (e.g., motor-impaired users). Our approach, in contrast, focuses on providing GUIs for process-oriented applications (e.g., booking applications), following the gender-inclusive design principle, i.e., we aim at generating GUIs that are equally well usable for all users. In fact, SUPPLE is a *user-centered* GUI generation approach, whereas the approach proposed in this dissertation is *usage-centered*.

We conjecture that customization is inevitable for achieving a GUI with a good level of usability and instead of focusing on identifying the "best" cost function configuration in terms of usability, we focused on providing further customization possibilities together with a tool-supported process for creating customized applications with UCP. Both are presented below.

5 Interaction Model Development and GUI Customization for Multiple Devices

The development of interactive applications with high-quality UIs is typically an exploration of design alternatives from which the most appropriate one for specific circumstances is chosen [PRS11]. More complex applications are not only developed in an iterative, but also incremental way. Our new interactive GUI generation approach supports four different ways of exploring design alternatives at different levels of abstraction in an iterative and incremental way.

First, we support the exploration of interaction design alternatives through our tool-supported process for iterative and incremental interaction model development using automated GUI generation. A high-quality interaction model is crucial for achieving GUIs with a good level of usability, because the interaction model specifies the overall behavior of the application and provides the basis for all GUIs that are derived for different devices.

Second, we already provide four different tailoring strategies to facilitate the exploration of alternatives. In addition, we provide the possibility for customizing the parameters of the cost function for customizing the way the GUI is tailored.

Third, we provide a set of *generic-custom transformation rules*. Their name seems contradictory, but these custom rules do not depend on a specific DoD Model or Discourse Model structure. These rules are more generic than the application-specific custom rules, which facilitates their reuse. Customization through such transformation rules provides an alternative and more predictable way to influence the tailoring process and to customize the resulting GUI than configuring the cost function. Such generic-custom rules explicitly match a specific Communication Model pattern, rendering the corresponding GUI according to their LHS and are not replaceable (e.g., through *basic* rules). Such rules can be used, for example, to explicitly split a relation or not. Thus, they make the resulting GUI more predictable while reducing the number of possible GUIs at the same time. Further, customization is possible through adding application-specific custom rules that transform the concepts of a specific domain (i.e., Domain-of-Discourse objects) in a more adequate way than the corresponding *basic* rule(s). In this sense, custom transformation rules can be used to explore layout, style and rendering alternatives (i.e., which attributes of a specific DoD object are rendered).

Fourth, we potentially support layout and style customizations of a specific Structural Screen Model through direct manipulation the Graphical Screen Model Editor (GSME) [Arm14]. These

customizations are made persistent for regeneration in the next development cycle through transferring them to a style sheet, if possible, or through storing them in a so-called *Change Model*. Thus the GSME also allows for exploring layout and style alternatives.

Our support for exploring design alternatives in these four ways allows us to support the customization of all GUI aspects that are typically distinguished in literature [Sze96, HME11, PHKB12]. The left column of Table 5.1 lists these aspects, and the right column shows the corresponding artifacts in UCP:UI that need to be adapted in order to customize a specific GUI aspect.

Table 5.1: GUI Aspects and Customizable UCP:UI Artifacts

GUI Aspects	**Supported through customization of**
1. Presentation Units / Navigation	Communication Model, Tailoring Strategy, Transformation Rules
2. GUI Elements and Properties	Transformation Rules, Screen Model
3. Clarification Dialog	Communication Model
4. Layout	Transformation Rules, Screen Model
5. Visual Appearance	Transformation Rules, Screen Model

Presentation Units / Navigation

The Presentation Units (PU) and the navigation between them are defined through the Communication Model. However, such PUs can be split through alternative transformation rules in the course of the Screen Model generation. This can either be caused through the tailoring strategy when the GUI is automatically tailored for a specific device, or through the designer when manually defining and matching specific transformation rules.

GUI Elements and Properties

All GUI elements (i.e., widgets) that populate the Structural Screen Model are created through the RHS of a transformation rule. Their properties are specified in a corresponding Cascading Style Sheet (CSS) and are already considered during the GUI generation. Specific widgets can only be defined in the RHS of a specific transformation rule. Their properties in contrast can either be specified upfront the application of the transformation rule in the corresponding CSS file or they can potentially also be customized through editing the property of a specific Screen Model widget in the GSME.

Clarification Dialog

In principle, clarification dialogs involve interaction between the user and the system and thus have to be modeled explicitly in the Communication Model. However, we use a simple form of input verification to generate simple clarification dialogues automatically. In particular, we check whether a specific attribute (represented through an input field in the GUI) has been defined as compulsory in the Domain-of-Discourse model. Before submitting the corresponding screen, we check whether all such attributes have been provided by the user. If not, we do not submit the form but display an automatically generated pop-up dialog that clarifies for the user which attributes still need to be provided to proceed.

Layout

The layout of a GUI screen is partially composed of the layout specified in the corresponding transformation rules' RHSs and completed through the layout module. Hence, it can either be customized in the RHS of a transformation rule (either explicitly or using our Layout Hints), or it can be customized for a specific screen in the Screen Model (i.e., after it has been completed through the layout module).

Visual Appearance

The visual appearance of a specific widget is defined through properties specified in the corresponding style sheet.

In the remainder of this Chapter, we present our support for customizing these five GUI aspects in detail. In particular, we present an iterative process that supports high-quality interaction model development first. Then we extend this process to support incremental interaction model development, which facilitates the development of more complex applications. These processes allow for supporting GUI aspects 1 and 3 through customization of the Communication Model and selection of the tailoring strategy. However, further customizations are required to achieve a good level of usability through the desired "look & feel". So, we extend our iterative and incremental process to support GUI customization through specific transformation rules already during the development of the interaction model. Such, specific transformation rules support the customization of GUI aspects 1, 2, 4 and 5. Alternatively, we support the customization of the aspects 2, 4 and 5 through customization of the Screen Model. We will shortly introduce the GSME tool [Arm14] (developed by Alexander Armbruster) and introduce the Change Model that allows for persisting modifications performed in the GSME that cannot be persisted through a style sheet. Finally, we show how customizations that were performed for the GUI of a specific device can be (partly) reused when customizing the GUI for another device. In fact, facilitating GUI customization for multiple devices.

5.1 Iterative Interaction Model Development

A GUI is defined through its structure and its behavior. In model-driven GUI generation the behavior is typically specified through the high-level interaction model and the structure is derived through model-transformations. It is vital to develop high-quality interaction models for achieving high-quality GUIs, because the interaction model is the very same for all GUIs of a certain application. High-quality interaction models are typically achieved through iterative development.

Let us illustrate the need for iterative interaction design with a case study that we conducted with a small bike rental application [RWP+13]. We explicitly considered usability heuristics and guidelines while developing the interaction model, and while developing our *basic* transformation rule set. Subsequently, we generated a desktop and a smartphone GUI fully automatically and four usability experts performed a Heuristic Evaluation based on the same set of heuristics and guidelines, to locate the shortcomings of automated GUI generation. About 40% of the overall violations on both devices resulted from an inadequate Communication Model, which means that improving the interaction model in at least one more iterations was inevitable to achieve a high level of usability.

In general, there is a vast body of literature on interaction design [Bro93, SPAS97, vSvdVB97], which offers guidelines [CP09] or patterns [Bor00] for different domains (e.g., recommender systems [SS02]). However, interaction design is an iterative process, where alternative ways to achieve a certain interaction need to be explored and compared to find the most suitable one for a given context [PRS11]. For example, providing additional information for adequate user guidance is important to achieve a high level of usability, but which information is exactly required is hard to estimate without user tests [FCDC10, GFCDCM13].

So, we propose to use automated GUI generation to improve an interaction design [RKP+14]. The key idea is to facilitate the exploration of alternative designs through making derived GUIs available quickly and cheaply through the generation. These generated GUIs facilitate the evaluation of the corresponding interaction designs through their "look & feel". Note the difference to an earlier approach for perfect fidelity prototyping [FPR+07] with its direct focus on improving the generated GUIs. Our new approach focuses on support for improving the interaction design first, and makes use of a stable version for concurrent improvement of the GUIs (front-ends) for various devices and the development of the application logic (back-end).

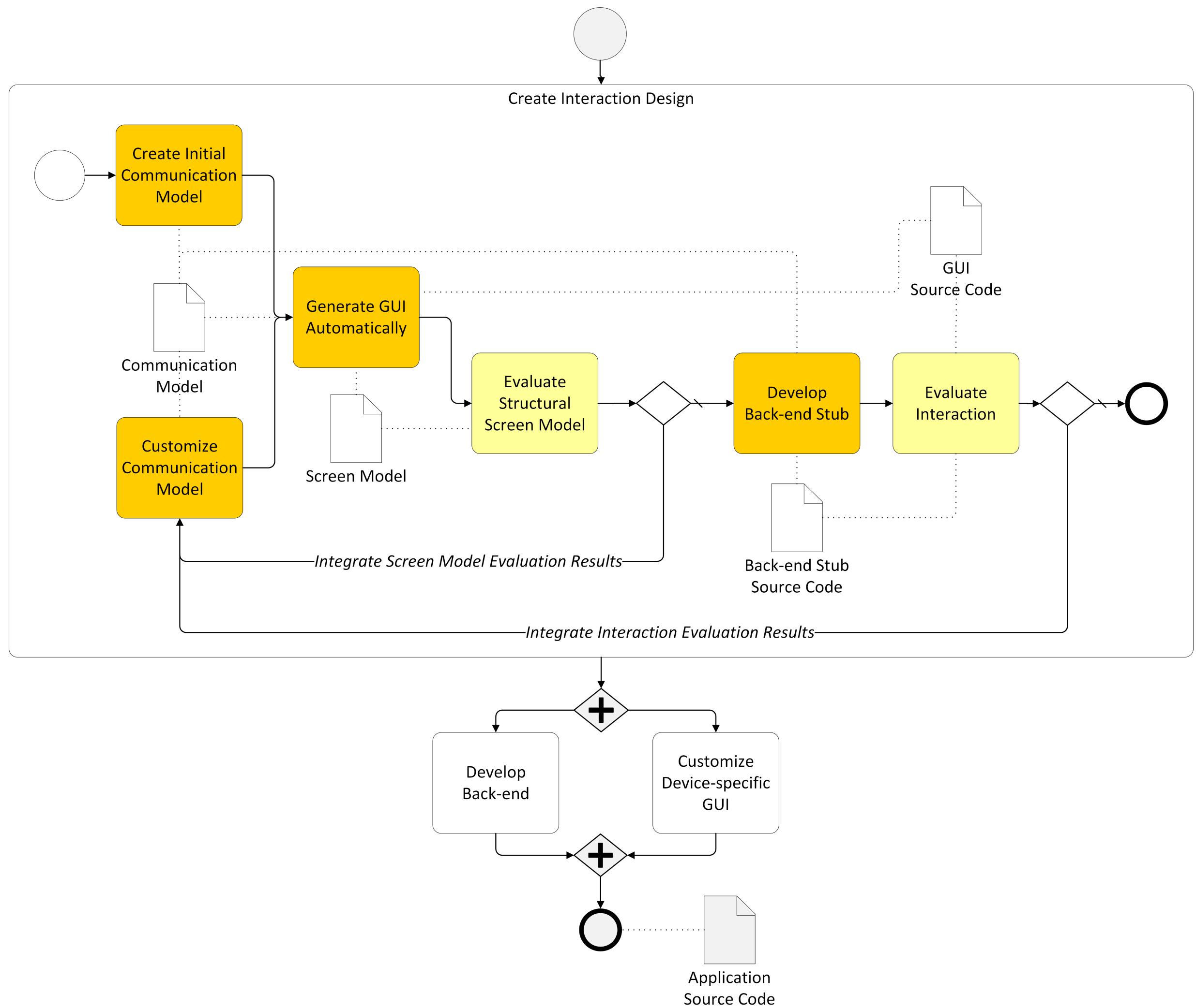


Figure 5.1: Our Tool-supported Process for Iterative Interaction Design Embedded in our Communication Model-based Application Development Process [RKP+14]

Figure 5.1 shows our Communication Model-based application development process in BPMN (Business Process Model and Notation).[1]It includes and starts with the tool-supported process for iterative interaction design named *Create Interaction Design*. The concurrent activities *Develop Back-end* and *Customize Device-specific GUI* follow, which result in the `Application Source Code`. These two activities are presented below and are out of scope for this section, since they do not contribute to developing the interaction design. Rather, a stable interaction design is already assumed here for performing these activities concurrently, because the Communication Model that specifies the interaction design also specifies the interface between back-end and GUI in UCP [PR11].

In Figure 5.1, all activities shown in orange (dark) result in new or customized artifacts, while all activities shown in yellow (light) are evaluation activities. Each of these activities is a special case of one of the generic activities designing alternatives, prototyping and evaluating as involved in interaction design according to [PRS11]. UCP offers tool support for a human designer for all these activities, which is publicly available on the UCP update site.[2]

We distinguish between a *Create Initial Communication Model* and a *Customize Communication Model* activity, since they are inherently different. While each of them results in a `Communication Model` artifact, creating a first version from scratch requires more creativity than customizing according to results from an evaluation. Once a Communication Model is available, it can be transformed in the activity *Generate GUI Automatically* to a `Screen Model` and the `GUI Source Code` fully automatically using UCP. The Structural Screen Model represents the GUI on the Concrete UI Level. It is comparable to GUI mockups and can, therefore, be used for a first evaluation in the *Evaluate Structural Screen Model* activity. These evaluation results can be immediately used to customize the Communication Model by following the path labeled *Integrate Screen Model Evaluation Results* in Figure 5.1. This allows iterating on design alternatives in micro-iterations.

Once the results from the evaluation of the screen model are satisfactory, our process continues with the *Develop Back-end Stub* activity. It results in a prototypical application back-end stub, as required to achieve a running application prototype. It can be used in the *Evaluate Interaction* activity, where the focus is on its external behavior in the course of interaction with a user. The results of this evaluation can be used to customize the Communication Model. In effect, this allows iterating on design alternatives in macro-iterations.

Now let us sketch the six activities of this process.

Create Initial Communication Model

This activity creates an initial Discourse-based Communication Model from scratch, while parts from previously generated and evaluated models may be reused here for recurring interactions. It represents all possible flows of interaction between the user and the application, domain concepts communicated about in this course, and actions that can be performed by either of the interacting parties. This activity precedes all other activities and is not part of the customization iterations, because the Communication Model needs to be created from scratch only once.

Creating a Communication Model involves creating a Domain-of-Discourse (DoD) Model, an Action-Notification Model (ANM) and a Discourse Model. We provide a wizard for this initial

[1] `http://www.omg.org/spec/BPMN/2.0`
[2] `http://ucp.ict.tuwien.ac.at/eclipse-update/`

Communication Model creation, which collects the required information from the interaction designer and subsequently creates the corresponding model files and launch configurations for the final application. In particular, it creates empty model files for an application specific DoD Model, an ANM and a Discourse Model. It links the respective model files correctly, adding also references to the *basic* ANM [Pop12], and the Ecore meta-model as DoD Model [Š14].

UCP:UI is based on the Eclipse Modeling Framework[3] (EMF) and specifies the DoD Model in Ecore, which is the Eclipse implementation of the Meta Object Facility (MOF) standard. The Ecore meta-model already provides the basis data types (e.g., Integer, String or Object) and is supported through a graphical editor, which can be used by the designer to specify additionally required concepts in the DoD Model. It is sensible to specify a dedicated DoD Model for an application, because this model specifies the concepts that the two interacting parties (i.e., user and system/application) can "talk about" and is not intended to model a certain domain. DoD Model concepts are typically less detailed than domain ontology concepts. For example, there is no need to model the number of gears in a bike rental DoD model, if all bikes are identical, but the DoD Model concepts can of course be mapped to elements of an existing domain ontology by the back-end.

Each application will most probably require dedicated Actions or Notifications. These can be specified in the ANM, for which we provide a graphical editor [Pop12]. Comparable to the Ecore meta-model, we already provide an ANM that specifies *basic* Actions like GET, SET or SELECT and Notifications like PRESENTING, which have been required in all applications that we built so far.

The Discourse Model specifies all possible flows of interaction between the user and the system for a given application. We typically specify use cases for the application to build and detail their specification using the Discourse Model, which in fact is a Discourse-based Communication Model, if the Propositional Content of the Communicative Acts that references concepts of the DoD Model and the ANM, is specified. UCP:UI provides a graphical editor [FKP+09] for the creation and customization of Discourse-based Communication Models.

The creation of an initial interaction design can be based on design patterns or alternative interaction designs as presented in [RPK13a]. We additionally found that the following four best practices (BPs) were useful to achieve an initial Communication Model.

- **BP1 – Start with the Discourse Model** and create the DoD Model and the ANM subsequently.
- **BP2 – Assume a potentially infinite screen.**
- **BP3 – Match the granularity of the Domain-of-Discourse Model concepts to Communicative Acts.**
- **BP4 – Provide adequate names and descriptions for Discourse-Model elements.**

Best practice number 1 and 2 facilitate focusing on the interaction design, instead of domain or device specific application aspects. Number 3 results in GUIs with a basic level of usability, even when only the *basic* rules provided by our transformation tool are available. The reason is that these rules create widgets for all DoD concept attributes, but do not resolve references. Number 4 facilitates the evaluation of the Structural Screen Model and the interaction through the running application, because Discourse-Model elements names and descriptions are used as label texts by the *basic* transformation rules.

[3] `http://www.eclipse.org/modeling/emf/`

BP1 – Start with the Discourse Model. Starting with modeling the interaction has the advantage that the designer can concentrate on specifying the flows of interaction rather than on what exactly is exchanged. This way she does not have to keep a certain DoD Model or ANM in mind, or has to adapt these models subsequently, but can develop them according to the Discourse Model when specifying the Propositional Content of the Communicative Acts. All three models may have to be adapted in subsequent iterations based on the feedback of the GUI evaluation / application test. Starting with the DoD Model instead involves the difficulty that designers have to anticipate the interaction and typically start modeling the application domain rather than concentrating on which domain concepts are required for interaction. We do not recommend to start with the ANM for the same reason.

BP2 – Assume a potentially infinite screen. Interaction models must not consider any device constraint if they provide the common basis for GUIs for different devices [RMB13]. In addition they need to model all information that can be exchanged at a certain point in time needs as concurrently available (see Subsection 2.1.3). It is helpful to imagine a potentially infinite screen and to start rendering the GUIs with the one for the largest screen, ending with the smallest screen.

BP3 – Match the granularity of the DoD Model concepts to Communicative Acts. The DoD Model is not a domain model, hence it is no problem to adapt the granularity of the DoD Model concepts to the granularity of the Discourse Model. The granularity of the Discourse Model are Communicative Acts. By matching this granularity with the DoD Model granularity we mean that each Communicative Act should reference classes of the DoD model. The reason in this case has a technical background. Our *basic* rule set renders only the attributes of content objects, if they are classes, or the content object itself, if it is a basic data type like Integer or String. Objects that are referenced from a given content object are ignored by our default rule set for two reasons.

First, the referenced object might reference other objects, which leads to a densely populated GUI if the attributes of all referenced objects are rendered (there could be even circular dependencies), and according to our experience there are only a few attributes of referenced objects that are relevant in addition to a specific content object's attribute. For example, after having logged-in we want to inform the user that she (her name is Julia) is logged in by displaying a “Hello, Julia”. This can be modeled through an Informing Communicative Act whose content object is of the type User, as modeled in the Bike Rental DoD Model shown in Figure 2.2. In this case you would not even want to display the User's password attribute, not to talk about the attributes of the referenced CreditCard, Person and Address class. Rendering only the username attribute can be achieved through a custom transformation rule, which will be presented in detail in Section 5.3.

Second, the content objects are serialized for the transmission between the GUI and the application [PR11]. Content objects that reference other objects need to model this references as containment (i.e., UML aggregation), the referenced object will not be serialized otherwise. Objects referenced through a containment reference can of course be rendered too, again through specifying a custom transformation rule.

BP4 – Provide adequate descriptions and names for Discourse Model elements. The Propositional Content of a Communicative Act is formally specified in an SQL-like language (see Subsection 2.3.1). Furthermore, it provides the possibility to describe the content in natural language. These descriptions are used as default values for heading labels by the *basic* transformation rules. This rule set uses the names of Discourse Relations and Adjacency Pairs for naming the corresponding containers that are created for them (e.g., a tab in a tabbed panel). Providing human readable descriptions and names improves the usability (i.e., the wording) of the automatically generated GUIs and facilitates establishing the connection between GUI screen and Communication Model elements for the human designer when testing the application. The wording can of course be adapted for each device, when the corresponding Screen Model is customized (see Section 5.4).

Generate GUI Automatically

This activity is essentially a machine task performed fully-automatically by UCP as presented in [PRK13]. It generates the `Screen Model` (as an intermediate result) and the `GUI Source Code`.

Rendering a GUI for a specific Communication Model requires a device specification and the selection of a tailoring strategy. Our wizard creates a default rendering configuration, for each device that has been selected which provides default values for the device specification (a desktop device) and the tailoring strategy (Screen-based Device Tailoring with Vertical Scrolling) along with the Communication Model [Š14].

UCP:UI provides the three default device specifications – Desktop PC, Tablet PC and Smartphone (see Table 5.2). Interaction model development as a basis for multi-device GUI development should not consider device properties at all. We conjecture it suffices to roughly distinguish these three device classes for the development of the interaction model, and preferably render the GUI for the desktop to avoid violating our optimization objectives and keep the required automated tailoring loops minimal.

Table 5.2: Selected Default Device Specification Properties and Tailoring Strategy

Device	Resolution (x,y), dpi	Pointing Granularity	Toolkit	Tailoring Strategy
Desktop[4]	1920x1200, 72 dpi	fine	HTML	no scrolling
Tablet[5]	1280x800, 96 dpi	coarse	HTML	no scrolling
Smartphone[6]	480x360, 96 dpi	coarse	HTML	vertical scrolling

The designer can alternatively use her own custom device specifications, if already available, while prototyping the interaction. We provide a graphical tree editor to create/customize device specifications and we also provide tool support for creating new launch configurations, respectively accessing and customizing them [PRK13].

Figure 5.2 shows the GUI that we provide for configuring the rendering process. The view of the "General" tab visible in this figure allows for the specification of the Communication Model to transform, which transformation steps to execute, where to save the intermediate (Behavioral

[4]e.g., Desktop PC with Dell E2484WFP Monitor
[5]e.g., Samsung Google Nexus 10 Tablet (with scaling factor 2 for the resolution)
[6]e.g., Apple iPhone up to 4S (with scaling factor 2 for the resolution)

and Structural) Screen Model and where to find the implementation classes for the Domain-of-Discourse Model. The view of the“Discourse –>Structural UI” tab allows for providing the device-specification and application-specific transformation rules, as well as the selection of a tailoring strategy. The view of the “UI Code Generation” tab allows for the selection of the target-toolkit (e.g., HTML), the specification of an application-specific style sheet, base package name and source folder for the final GUI code. The view of the “Runtime” tab allows for the specification of the partner service that represents the application logic, and the view of the “Common” tab allows saving the launch configuration.

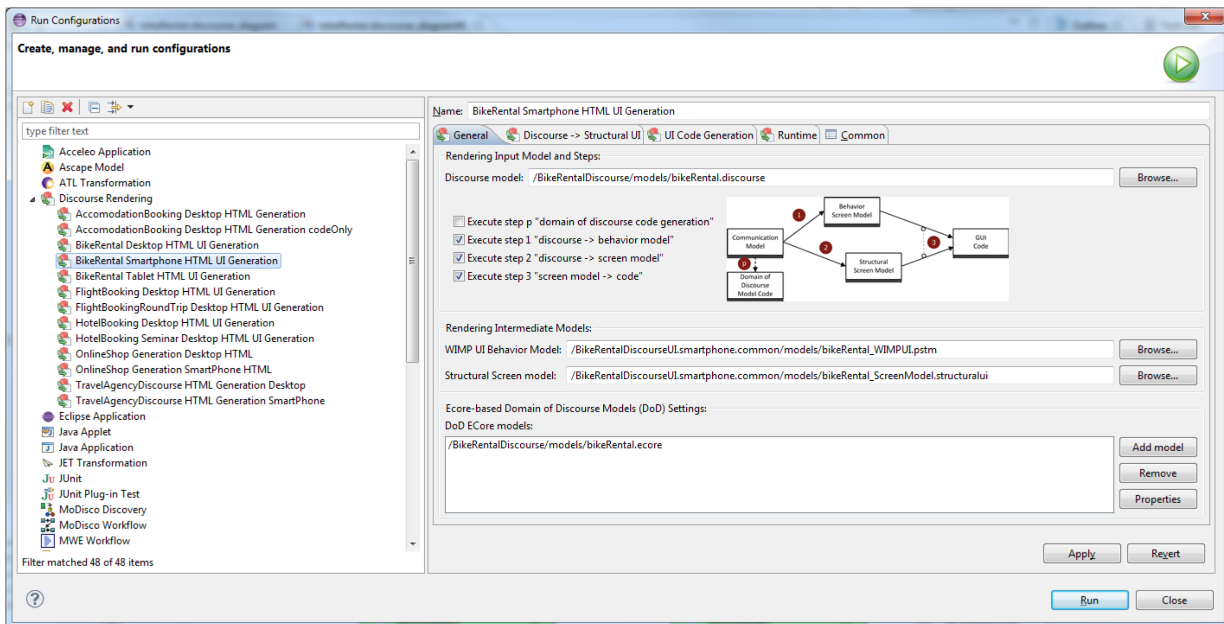


Figure 5.2: Rendering Launch Configuration [PRK13]

Furthermore we provide rendering support in form of model integrity checks, to make the transformation process more robust and to facilitate the detection and localization of modeling errors [Sch10, RSKF11]. Using the EMF Validation Framework formulate and execute the integrity checks, we distinguish three different types of integrity checks, based on the following ways of implementing constraints: Model Constraints, OCL Constraints and Java Constraints. Applying all these three types of integrity checks allows us to restrict the model space as shown in Figure 5.3.

We provide and apply such integrity checks for the Communication Model, the Transformation Rules, the Structural and the Behavior Screen Model and trigger the next transformation step only if a model is compliant our integrity checks. In case of an error the transformation process designer is informed about the source and nature of the problem and the generation process is stopped.

Evaluate Structural Screen Model

This activity evaluates the screens of the Structural Screen Model, which are comparable to GUI mockups. It allows identifying missing interaction elements and thus shortcomings of the interaction model through comparing the generated GUI model to the functional requirements (e.g., given through use cases) that the resulting application needs to satisfy. Importantly, this is

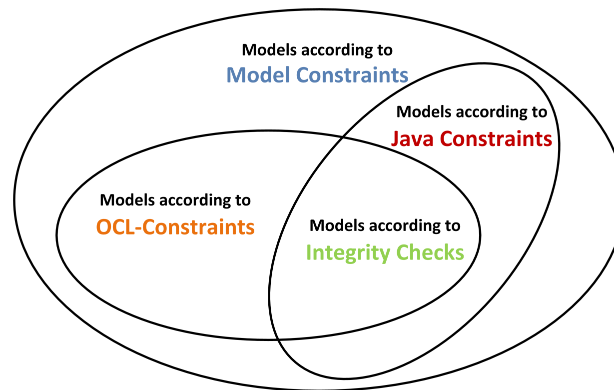


Figure 5.3: Model-space Restrictions through Integrity Checks [RSKF11]

feasible even without the availability of an application back-end (prototype or final). Therefore, it allows for micro-iterations in our process.

UCP provides a dedicated graphical editor for the Structural Screen Model [Arm14] that supports this evaluation activity through visualization of the generated screens.

Develop Back-end Stub

This activity provides a stub for the back-end that satisfies the functional interface between the generated GUI and the application logic, which is defined through the Communication Model [PR11]. The resulting artifact is the `Back-end Stub Source Code`, which represents a minimal prototypical application back-end. This stub and the GUI together provide a first running prototype.

In particular, the UCP run-time module (UCP:RT) uses the Communication Model as service specification and implements a Service Oriented Architecture (SOA) [Pop12]. The application back-end is so-called "machine service", as opposed to the GUI which is a so-called "user service". The functionality of such a machine service is specified through the Propositional Content of the Communicative Acts and through conditions specified for branches of Discourse Relations. UCP:RT implements an MVC architecture [PKR13] and requires the back-end developer to implement the functionality related to the calls in a so-called Application Adapter. UCP includes a generator for the source code of the Application Adapter [Š14]. The developer has to complete this code skeleton by implementing the stub, which needs to provide sample data and handle the internal state of the application back-end as specified in the Communication Model.

The full implementation of the back-end typically commences in parallel to the iterative customization of the GUI as soon as the Communication Model has been tested sufficiently and is not going to be changed any more. A stable Communication Model is the pre-requisite to decouple the back-end development and the GUI customization, as the Communication Model specifies the interface for both services.

Evaluate Interaction

This activity uses the running application prototype to evaluate the interaction. This may be done informally by the developer, again through comparing the interaction to the functional

requirements for the application, but is typically achieved through a Heuristic Evaluation performed by usability experts, or possibly even through usability tests, but always with a focus on the interaction rather than the GUI screens per se. Only problems or violations of heuristics concerning the external behavior are relevant, because layout and style are not reflected in the interaction design.

Customize Communication Model

This activity customizes the `Communication Model` representing the current version of the interaction design according to the results of the evaluation activities. Whatever flaw or problem they have revealed, another design alternative intends to fix or resolve it. This may entail changes of the flows of interaction between the user and the application, of the model of domain concepts communicated about in this course, and of actions that can be performed by either of the interacting parties. Of course, also combinations of such changes may be necessary.

The Create Interaction Design activity, that contains our iterative interaction development process, is finished as soon as no more adaptions are required. This process has been applied during the development of a small Bike Rental application, which is presented in detail in Section 6.1.

5.2 Iterative and Incremental Interaction Model Development

Engineering more complex applications includes more complex interaction. Such complex applications are typically developed in an iterative and incremental manner. We, therefore, extended our process for iterative interaction development presented above, to support incremental development as well [RPK+14]. Our tool-supported process for iterative and incremental interaction design in BPMN[7] notation is shown in Figure 5.4. Similar to our iterative development process, it includes and starts with the tool-supported process named *Create Interaction Design*. Once a stable interaction design is available the activities *Develop Complete Back-end* and *Customize Device-specific GUI* can again be performed concurrently as they are independent of each other. Again, these two activities are presented below and are out of scope for this section.

In Figure 5.4, all activities shown in orange (dark) result in new or customized artifacts, while all activities shown in yellow (light) are evaluation activities. Our iterative and incremental process starts with the *Create/Adapt Communication Model* activity. This activity creates the initial `Communication Model` and adapts it in subsequent iterations. Such Adaptations can either be modifications due to the results of an evaluation activity, or incremental extensions that introduce new functionality according to the requirements and may concern any of the three models that constitute the Communication Model (i.e., the Domain-of-Discourse, the Action-Notification or the Discourse Model).

Once a Communication Model is available, it can be transformed in the activity *Generate GUI Automatically* to a `Screen Model` and the `GUI Source Code` fully automatically using UCP. The Structural Screen Model can, again, be used for a first evaluation in the *Evaluate Structural Screen Model* activity. These evaluation results can be immediately used to customize the Communication Model by following the path labeled *Integrate Screen Model Evaluation Results* in Figure 5.4. This allows iterating on design alternatives in micro-iterations. Once the results from the evaluation of the screen model are satisfactory, our process continues with the *Develop/Adapt Back-end*

[7] `http://www.omg.org/spec/BPMN/2.0`

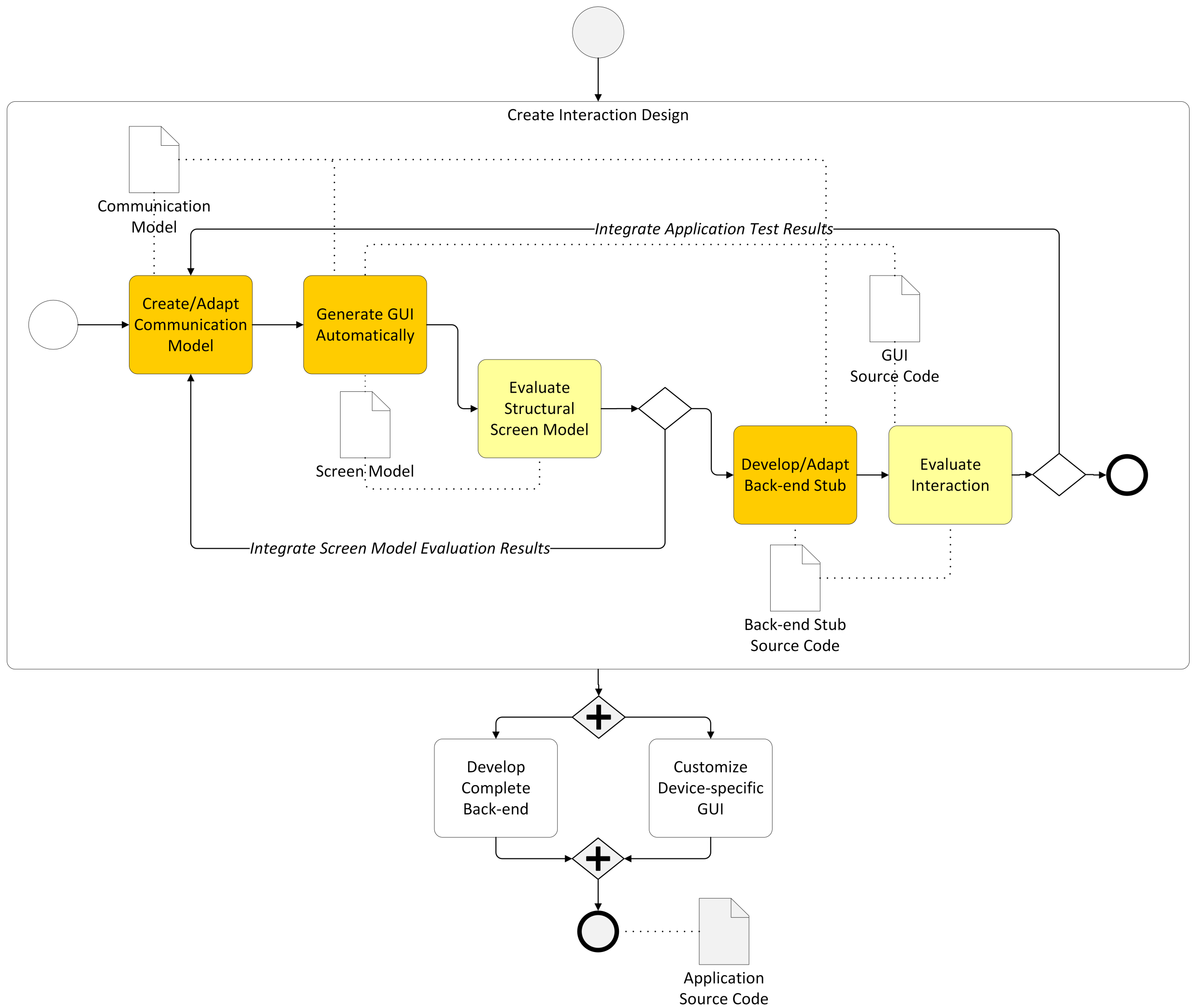


Figure 5.4: Our Tool-supported Process for Iterative and Incremental Interaction Design Embedded in our Communication Model-based Application Development Process [RPK+14]

Stub activity. Developing the back-end stub is the same activity as in our iterative process and means the initial creation of a back-end stub. Adapting the back-end stub can either be modifications due to the results of an evaluation activity, or incremental extensions that implement the new functionality introduced through an extension of the Communication Model. This activity results in a prototypical application back-end stub, as required to achieve a running application prototype. It can be used in the Evaluate Interaction activity, where the focus is on its external behavior in the course of interaction with a user. The results of this evaluation can be used to customize the Communication Model. In effect, this allows iterating on design alternatives in macro-iterations. Overall, this process is very similar to the iterative design process presented in the previous section, because each incremental extension is typically refined in subsequent iterative development cycles. To avoid redundant descriptions, we only presented the activities that are required to support incremental and iterative development in detail in this section. A detailed description of the *Generate GUI Automatically*, the *Evaluate Structural Screen Model* and the *Evaluate Interaction* activity has already been provided in Section 5.1.

As an empirical evaluation of our iterative and incremental interaction development process we present its application during the development of a more complex vacation planning application, which we present in Section 6.2.

5.3 Customization through Custom Transformation Rules

A high-quality interaction model is necessary as a basis for generating usable GUIs. However, it is typically not sufficient to achieve the desired GUI, in particular, the desired "look & feel". So, we extend our iterative and incremental process to support GUI customization through *custom transformation rules*.

A transformation rule supports the transformation of a specific Communication Model pattern to a specific Structural UI Model pattern. Custom transformation rules for a specific Communication Model provide the means to specify the corresponding custom GUI part in the rules's RHS. For example, displaying selected attributes of a DoD Model concept in a specific way (e.g., displaying the image of a hotel on the left side and its name and address on the right side). We found that applying GUI customization through transformation rules is already helpful during the iterative and incremental development of the interaction model and we, therefore, extended our interaction development process accordingly.

Figure 5.5 shows our extended iterative and incremental development process, which optionally allows for GUI customization through adapting the transformation during the interaction development. The activity *Adapt Transformation* allows the definition of *custom transformation rules*, which translate an application-specific Communication Model pattern into a specific Structural UI pattern. These *custom* rules together with the *basic* rules constitute the rule set that is used by the *Generate GUI Automatically* activity to generate the `Screen Model`. This rule set is represented through the `Transformation Rules` artifact in Figure 5.5. Their definition has been modeled as optional for the micro- and the macro-iterations in Figure 5.5 through X-OR gateways in the process model after each evaluation activity.

Custom transformation rules inhibit all additionally matching *basic* rules from being executed (firing) during the automated tailoring, and thus allow for predictable GUI customization. At the first glance it looks like a contradiction that we reduce the number of possible GUIs through increasing the number of transformation rules. The explanation is that the additional rules inhibit potentially matching alternative rules and, therefore, reduce the number of rules if they match a pattern that potentially can be matched by more than one rule.

Let us illustrate this search space reduction using the Vacation Planning Communication Model excerpt that we used for illustrating our search space generation in Section 4.4 (see Figure 4.13 above). Table 5.3 lists the five Communication Model patterns that are matched and the corresponding *basic* rules in column one and two, respectively. Additionally provided *custom* rules are listed in column three.

For a given customization to be implemented, we have to add one or more new transformation rules to the rule set. These new rules can be specializations of existing rules in the rule set. An example of such a customization is, that we only want to display specific attributes of a given object in a specific 'context'. The Domain-of-Discourse Model for our Communication Model excerpt in Figure 4.13 above, contains the `Article` concept with the attributes *title*, *shortdescription*, *description* and *picture*. This concept is referred to from the Communicative

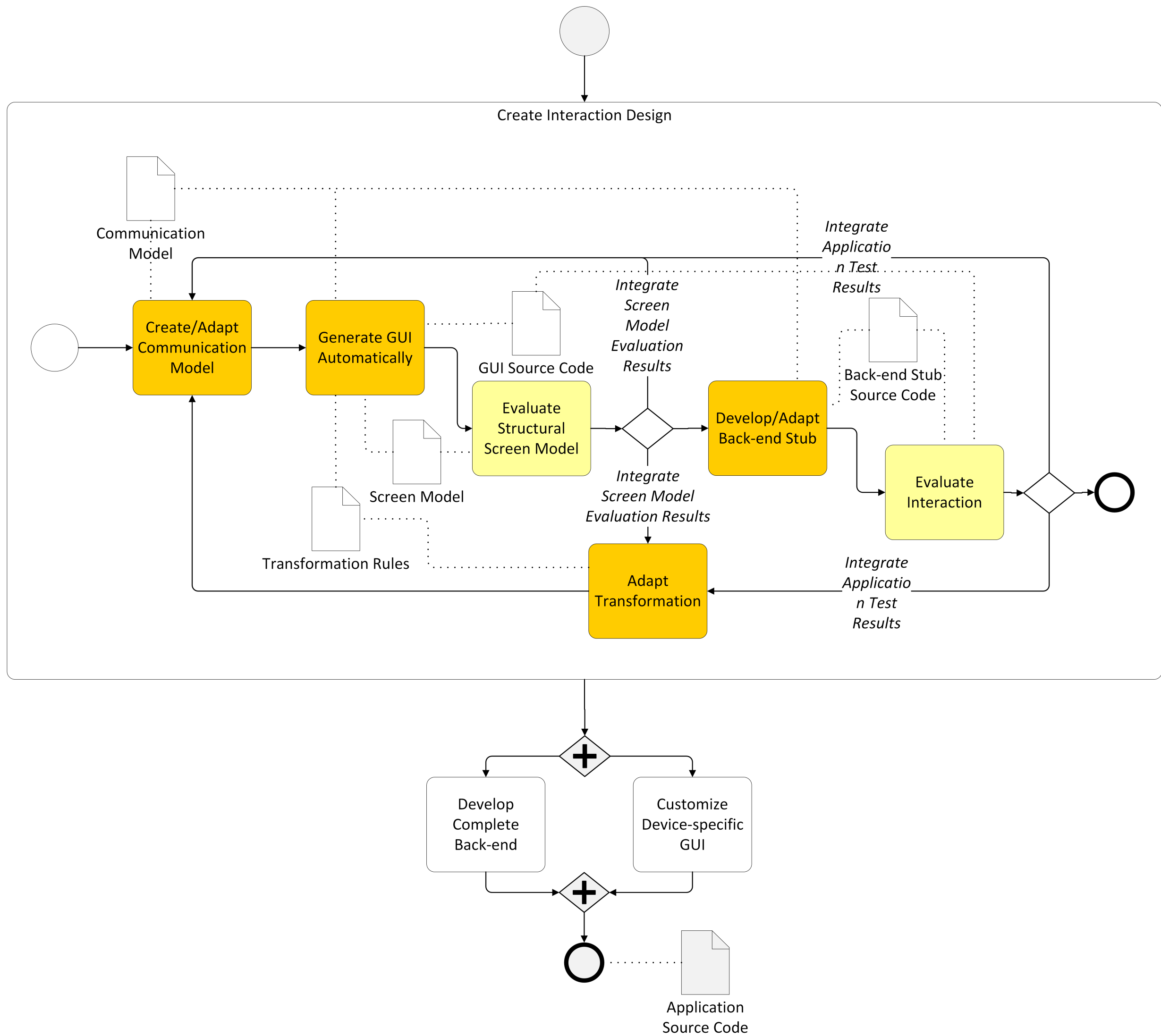


Figure 5.5: Our Tool-supported Process for Iterative and Incremental Interaction Design And Transformation Rule-based GUI Customization Embedded in our Communication Model-based Application Development Process

Acts in ADJ2, and the *basic* transformation rule creates widgets for all DoD concept attributes. For ADJ2, however, we only want to render the *title* and *shortdescription*. We achieve this through defining a custom transformation rule C2, which matches only ADJ2 and contains the desired customization in its right-hand-side (RHS).

In our example, there is now an additional rule available for matching ADJ2. This additional rule C2 extends the initial search tree (see Section 4.4) as shown in Figure 5.6 via dashed lines. In principle, adding new transformation rules may lead to a larger search space. This allows the generation of the customized GUI, but does not enforce it as desired by the designer.

To enforce the generation of such a customized GUI, we adapted the transformation engine in such a way that it supports the inhibition of rendering alternatives for Communication Model patterns matched by custom rules. For example, matching our custom rule C2 for ADJ2 inhibits executing (firing) the *basic* rules R5 and R6 for ADJ2. This actually reduces the search space as

Table 5.3: Transformation Rules for Vacation Planning Communication Model Excerpt

Pattern Identifier	Basic Rule(s)	Custom Rules
T1	R1	-
ADJ1	R2	C1
A1	R3, R4	-
ADJ2	R5, R6	C2
ADJ3	R5, R6	C3

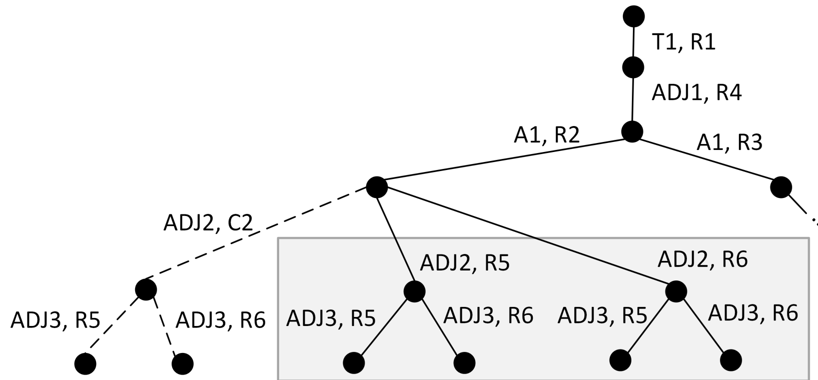


Figure 5.6: Search Tree for Vacation Planning Communication Model Excerpt

shown in Figure 5.6, by inhibiting the rendering alternatives in the gray rectangle.

This extension enforces the generation of a customized GUI through executing (firing) the corresponding custom transformation rules instead of all otherwise matching *basic* rules. This mechanism allows us to avoid adaptions of already existing transformation rules when new custom rules are added, because the *basic* rules are still matched, but inhibited and thus not executed (fired).

We added another custom rule C3 matching ADJ3 (rendering selected attributes of the DoD concept `Event`). Figure 5.7 illustrates the resulting search tree, which only contains two branches instead of the original eight as illustrated in Figure 4.14. Adding another custom rule C1 for ADJ1, as shown in Table 5.3, results in the generation of the start screen containing all desired customizations (presented in detail in Section 6.2). Note that custom rule C1 does not reduce the search space, because only one *basic* rule matches ADJ1.

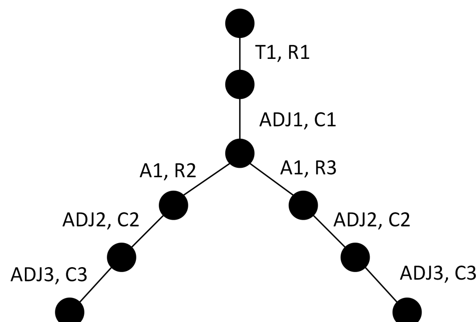


Figure 5.7: Customized Search Tree

To facilitate the customization through transformation rules, we provide a set of so-called *generic-custom transformation rules*. Our generic-custom rule set contains 17 transformation rules that

explicitly specify the name attribute of the Communication Model elements that they match (e.g., “Joint_split”), see Figure 5.8. Executing (firing) such a generic-custom rule for a specific Communication Model pattern can simply be achieved by naming the Communication Model elements correspondingly (e.g., naming a Joint relation “Joint_split”).

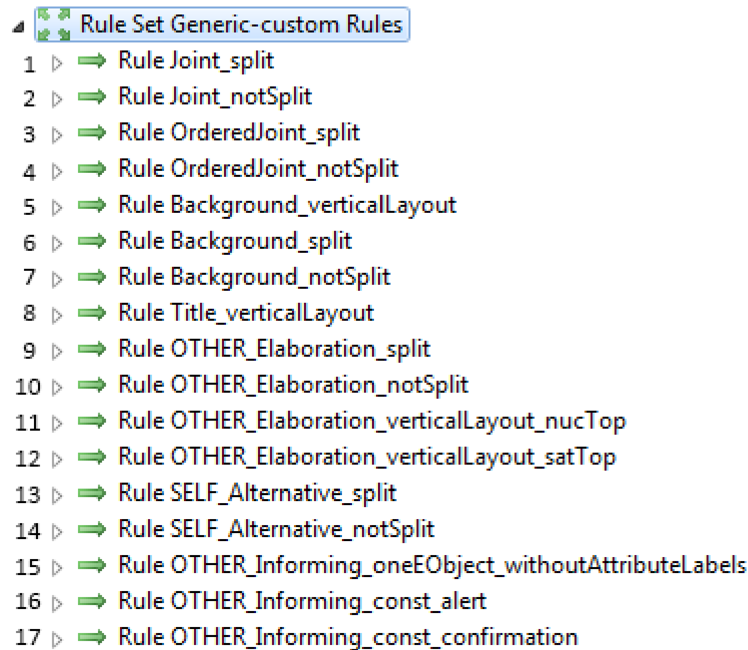


Figure 5.8: Generic-custom Transformation Rules

Our generic-custom set contains two rules for each relation that is splittable (i.e., Joint (1 & 2), OrderedJoint (3 & 4), and Background (6 & 7), Elaboration (9 & 10) and Alternative (13 & 14) assigned to the User) – one that *splits* the resulting Screen (similarly to a *basic alternative* transformation rule) and one that does *not split* the resulting Screen (similarly to a *basic* transformation rule). These ten rules can be used to reduce the number of alternatives and thus to speed up the tailoring process.

Moreover, we defined four rules that specify an alternative layout for Single Nucleus Relations (e.g., Background, Title or Elaboration) to *basic* transformation rules, in the rules' RHSs (rules 5, 8, 11, and 12 in Figure 5.8). These rules cannot be distinguished from the *basic* transformation rules by our cost function, as they create the same widgets as the corresponding *basic* rules. They can be matched manually by the designer if she prefers the layout over the one generated through the corresponding *basic* rule. The prefix *OTHER* or *SELF* in the rule names refers to the role that a particular relation or Opening Communicative Act of an Adjacency Pair has been assigned to. Only the Relations Joint, OrderedJoint, Background, Title and Sequence do not require the assignment of a specific role [Pop12]. When rendering a specific Communication Model, the Communication Model role selected in the launch configuration (typically representing the user) is mapped to SELF and allows selecting the appropriate rules automatically.

Our generic-custom transformation rule 15 provides a rendering alternative for Informing Communicative Acts that present a DoD concept of the type EObject. Our *basic* transformation rule renders a label containing an attribute's name and a widget that represents the corresponding value for each attribute of the object. This is helpful for the designer to connect the GUI parts to the corresponding Communication Model elements when rendering the first prototype. However,

some objects contain only attributes that are self explanatory (e.g., a picture), which makes the name labels obsolete in subsequent development cycles. Our generic-custom rule 15 provides such an alternative and creates only a widget for each attribute value.

Each transformation rule set ships with a corresponding style sheet that defines all styles that are referenced by RHS widgets. Our generic-custom rules 16 and 17 both create a label each for the Notification without a parameter presented by an Informing Communicative Act. The difference is that the label created by rule 16 references the `alert` style element, which renders the corresponding text in red, and rule 17 the `confirmation` style element, which renders the corresponding text in green.

In general, customization is device specific and requires a specific set of customization rules for each device. However, a custom rule set for a certain device may be reused and adapted for any other device, which facilitates the reusability of all customizations achieved through transformation rules.

5.4 Customization through Screen Model Adaptations

The Screen Model allows for device-specific, but still toolkit-independent customizations. It can be visualized through the Graphical Screen Model Editor (GSME) [Arm14]. This graphical representation can be used for a first evaluation while developing the interaction, as presented above. The GSME additionally supports GUI customization through direct manipulation of the Screen Model, which corresponds to the *Customize Device-specific GUI Model* activity in all three interaction development processes (see Figures 5.1, 5.4, and 5.5 above).

Figure 5.9 shows a screenshot of the GSME. The upper part of Figure 5.9 (delimited by a gray rounded rectangle) shows the CANVAS of the GMSE, containing the graphical representation of a screen from a small bike rental application, which we use to illustrate the layout customization capabilities of the GSME. This screen shows the previously selected bike and allows the user to confirm and rent or cancel the selected bike through the corresponding “Ok” and “Cancel” buttons. The lower part of Figure 5.9 (again, delimited by a gray rounded rectangle) shows the GSME’s PROPERTIES VIEW.

In particular, we support customization of the three GUI aspects *GUI Element Properties*, *Layout* and *Visual Appearance (Style)* on this level, as presented in Table 5.1 above. All these customizations can also be performed through specific transformation rules, whose RHS (i.e., Structural UI part) is also on the same level of abstraction (i.e., CUI level), which we conjecture is the most adequate level of abstraction. The difference lies in the interaction required to perform a customization. The transformation rule editor is an Eclipse-based tree editor, which allows for layout customizations, but does not visualize the layout directly. It also supports style customizations through referencing styles that can be defined in a corresponding style sheet, but visualizes them neither. The GSME, in contrast, supports *widget property*, *layout* and *style (i.e., visual appearance)* customizations through direct manipulation and visualizes them. We present all supported customizations for these three GUI aspects in detail below.

We do not support customizations of the GUI behavior or the creation of new GUI elements (i.e., widgets) through the Screen Model, because both require changes on the interaction model. The propagation of such changes to the interaction model would require support for round-trip engineering, including change propagation and model synchronization, which are research topics

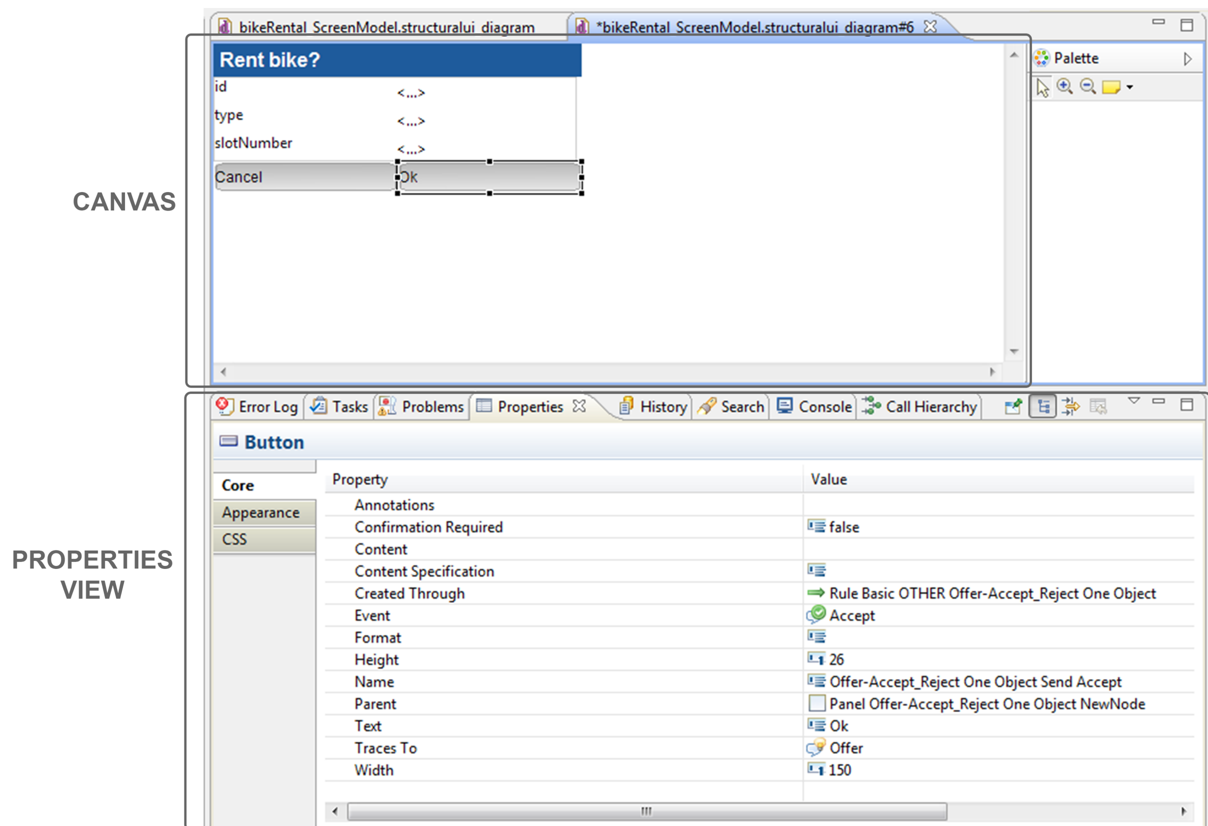


Figure 5.9: GSME Customizable Core Properties for *Static* Content

of their own and out of scope for this doctoral dissertation. However, we briefly discuss these problems and their future integration in UCP:UI in Chapter 8.

5.4.1 GUI Element Property Customization

The lower part of Figure 5.9 shows the GSME's *Core* page, which displays all properties of the "Ok" button selected in the CANVAS. The Core page allows for editing the *Content*, *Format*, *Text*, *Width* and *Height* properties. The *Content* property specifies the dynamic content (i.e., available at run-time) to be displayed by a specific widget. The *Format* property allows for the specification of a format String for such dynamic input or output. The *Text* property allows for the specification of static (i.e., already at design-time available) text that is to be displayed by a specific widget (e.g., "Ok"). Finally, the *Width* and *Height* properties specify the dimension of the widget. The content, format and text properties can only be customized through the Core page, the width and height property can alternatively be customized through selecting and resizing the widget in the CANVAS. The other widget properties listed by the Core page cannot be modified. The reason is that changing them would require further changes in an existing or the creation of a new transformation rule to be re-producible. We display them nevertheless, because they are helpful for debugging an application.

Let us illustrate the difference between static and dynamic content with the bike rental example. The screen shown in Figure 5.9 contains six widgets with static and three widgets with dynamic (i.e., at run-time available) text. The static text of the four labels (i.e., "Rent bike?", "id", "type"

and "slotNumber") and two buttons (i.e., "Cancel", "Ok") can be changed through selecting the corresponding widget in the CANVAS and changing the value of its text property in the Core page.

Dynamic content, in contrast, is only available at run-time (e.g., the previously selected bike) and is sketched through "<...>" placeholders in the GSME. Figure 5.10 shows the Core page for the label that will display the selected bike's id attribute in the running application. The dynamic value to be displayed by a specific widget, is specified through the widget's Content property. In Figure 5.10 the Content of the selected label is specified as id attribute of the type String. The containing class (i.e., the bike concept) is specified in the DoD Model and referenced in the Propositional Content of the Communicative Act specified in the widget's *Traces To* property. This property cannot be customized, but is helpful for debugging. Figure 5.10 also shows that the *Format* property of the Label is specified as String (according to the id type), and that no text is specified in the *Text* property, because the value to be displayed is not yet available.

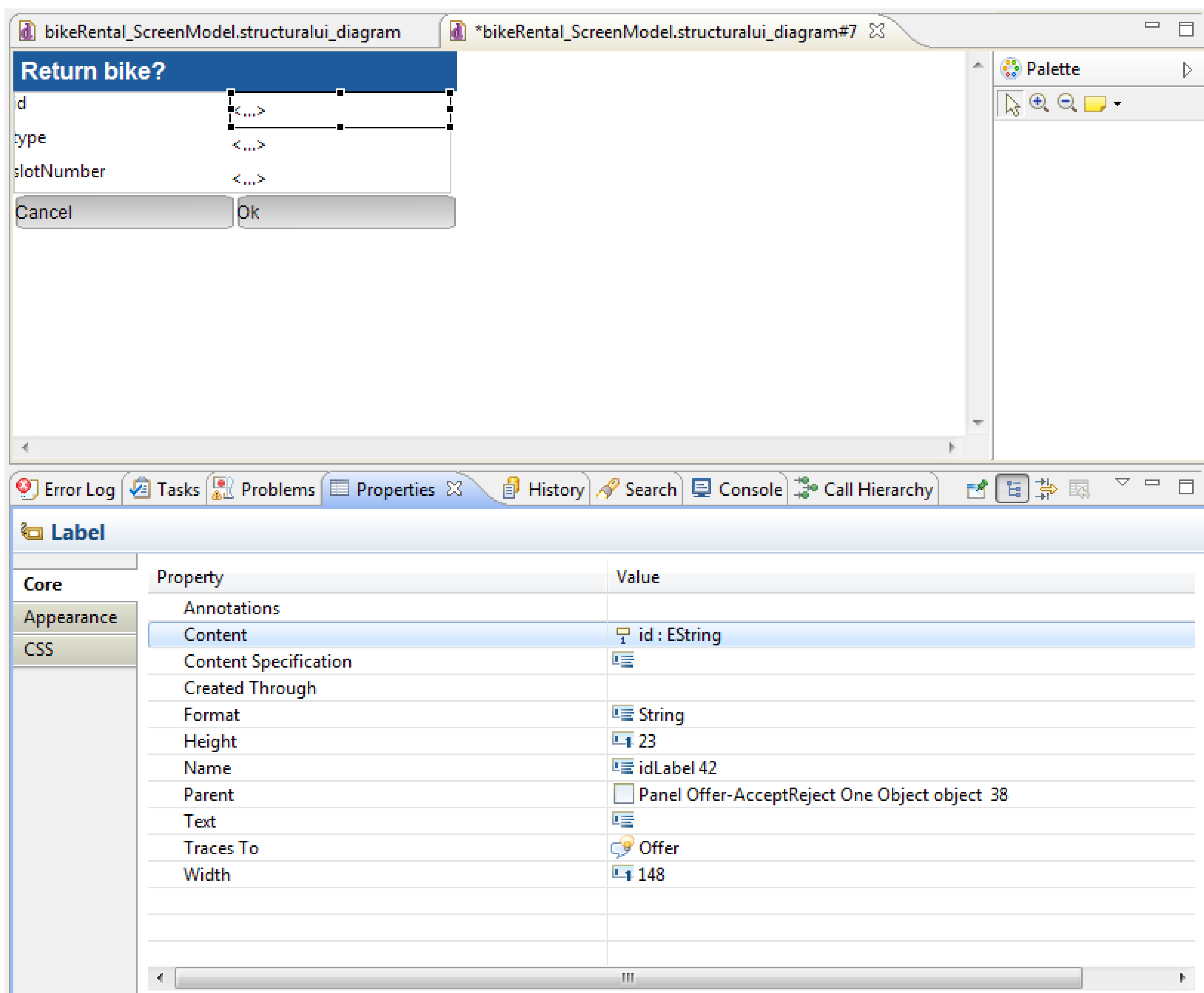


Figure 5.10: GSME Customizable Core Properties for *Dynamic* Content

All widget property customizations are stored for the subsequent source code generation through modifying the corresponding widget attribute.

5.4.2 Layout Customization

The layout of a specific screen can be customized in GSME through direct manipulation of the graphical representation on the CANVAS or manipulation of a specific widget through a menu. Direct manipulation allows for customizing a widget's position and size and menu manipulations allow for aligning a widget to the top, bottom, left, or right edge of its container, or to perform a re-size and an alignment operation of more than one selected widgets with a reference widget automatically. These automated re-size and alignment operations reduce the manual effort for layout customizations. The automated re-size operation allows re-sizing one or more selected widgets to the width, height or dimension of a reference widget. The automated alignment operation allows aligning one ore more selected widgets to a specific edge (i.e., top, bottom, left or right edge) of a reference widget. Finally, the GSME supports to automatically swap the position of two selected widgets. For more details on these operations please see [Arm14].

All layout customizations are stored for the subsequent source code generation through modifying the corresponding layout attributes.

5.4.3 Style Customization

Style customizations are supported through the *Appearance* and the *CSS* page in the GSME's PROPERTIES VIEW.

Figure 5.11 shows the *Appearance* page that allows for customizing the font properties *type*, *size*, *bold*, *italics*, *underlined*, *color* and the *background color* of the selected widget.

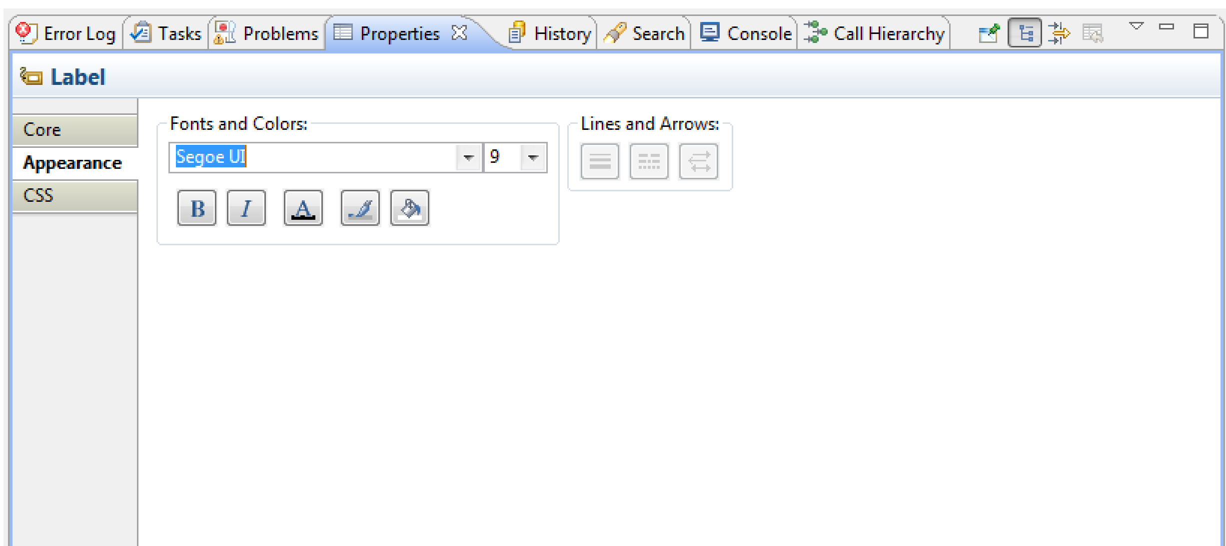


Figure 5.11: GSME Customizable Style Properties through Appearance Page

Figure 5.12 shows all additional style properties that can be modified through the GSME's *CSS* page. These properties correspond to the properties of the W3C Cascading Style Sheet (CSS) standard.[8]

[8]http://www.w3.org/Style/CSS/

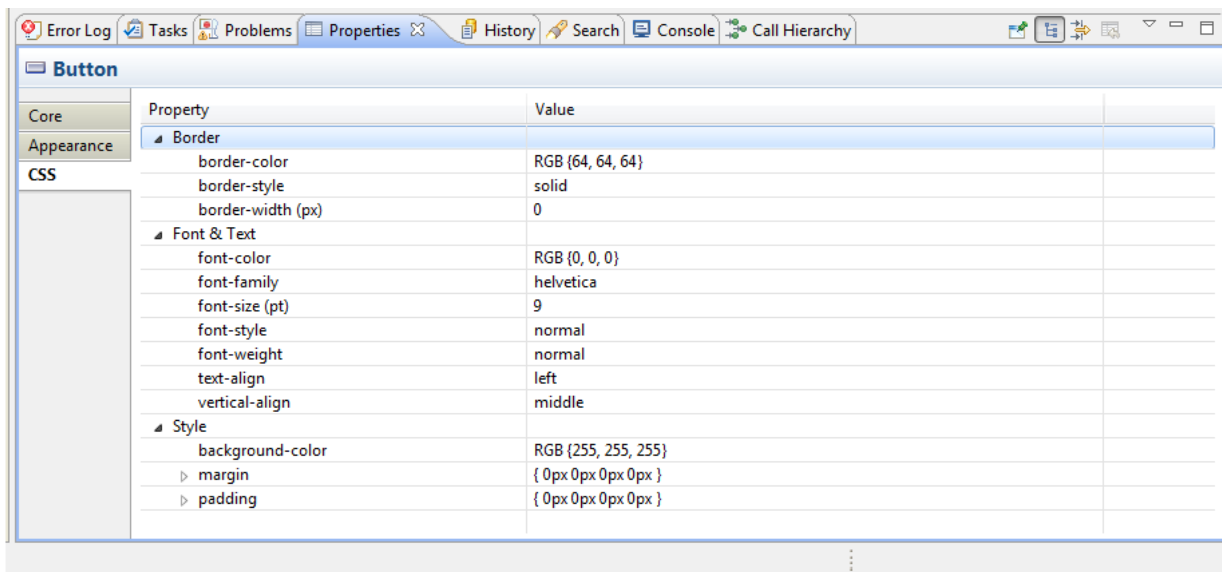


Figure 5.12: GSME Customizable Style Properties through Style Page

All style customizations are stored for the subsequent source code generation through creating a rule matching the widget id of the customized widget in a style sheet that is passed on to the code generation module.

The code generation module transforms the Customized Structural (and Behavioral) Screen Model to velocity[9] templates that are transformed to HTML pages at run-time. This module extracts all static text messages and stores them in a separate "messages" file [Ran08], which can be modified or replaced without having to regenerate the GUI and thus facilitates the customization of text messages or the adaption of the GUI to another language (i.e., localization).

5.4.4 Customization Persistence – The Changes Model

The CSS that stores the style customizations performed with the GSME can be provided as additional input to the rendering process. In case of re-generation, such customizations can already be considered in the automated GUI tailoring that generates the Screen Model through providing the corresponding CSS file as additional input. Widget property and layout customizations performed through the GSME, however, are stored in the corresponding widget and layout properties of the Screen Model and can only be included after the Screen Model has been generated. This implies that they have to be included anew after the Screen Model is available, in case of re-generation. To support this, we store such customizations in a separate model – the Changes Model – similar to the approach proposed by Pleuss et al. in [PWB13]. The idea is that in case of re-generation, the resulting Screen Model can be opened in the GSME and the changes can be re-applied. Technically, we want to trigger the GSME operations that were performed manually before, again to re-generate the corresponding customizations automatically.

Conceptually we based our customizations on [PVE+07], where beautification is defined as part of what we refer to as customization in this dissertation. A beautification is defined as a state-action pair (SAP) *state*, *action*, where a state corresponds to a widget and an action to the beautification

[9] http://velocity.apache.org/

operation.[10] We implemented this conceptual approach through constraining the customizations that can be performed through the GSME, which also executes the corresponding actions.

We modeled the information that needs to be stored for persisting a customization in the Changes meta-model, shown in Figure 5.13. In particular, this meta-model specifies all concepts that are required to store a GSME request. It does not model the customizations conceptually.

Its top-level class is a `ChangeLog` that allows for storing the Uniform Resource Identifier (URI) of the corresponding Screen Model in its `screenModelURI` attribute. Each customization is stored as `Change` and in particular, as either a `Property Change`, a `BoundsChange`, or a `ResizreOrMoveChange`. For each `Change`, we store whether it has been triggered manually or automatically as a consequence of a previous, manually triggered, change. The automatically triggered changes are only stored for logging, because they will be triggered again automatically when the previously performed manual changes are applied again.

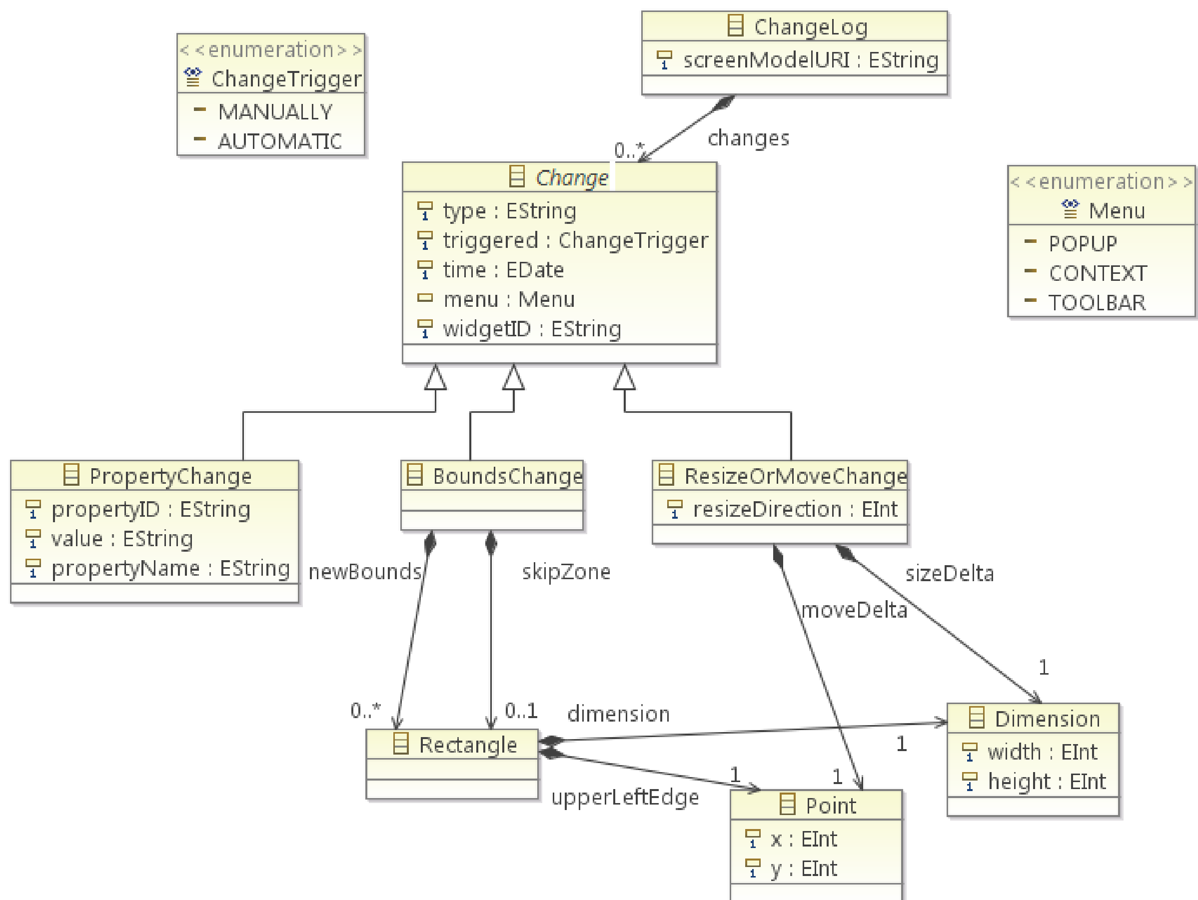


Figure 5.13: The Changes Meta-Model

A `PropertyChange` is created if a widget property is customized through the GSME. Such a change stores the `propertyID`, the new `value` and the `propertyName` in addition to the attributes of a `Change`.

[10]Customization in our context has a larger scope, as it includes, for example, also modifications of the Communication Model.

A `BoundsChange` is created if the bounds of one or more widgets are customized through the automated GSME re-size operation. It stores a list of `Rectangles`, defining the `newBounds` and an optional Rectangle as `skipZone`, which are required as input for the corresponding GSME operation. Each Rectangle in our Changes Model is defined through an `upperLeftEdge` of the type `Point` and a `dimension` of the type `Dimension`.

A `ResizeOrMoveChange` is created if a widget has been moved or re-sized manually by the GSME. It stores the `resizeDirection` (i.e., enlarge or shrink) encoded as Integer, a `moveDelta` of the type `Point` and a `sizeDelta` of the type `Dimension`.

Basically, the concept of providing additional customization information for a model is comparable to the transformation templates used in [AVVP09]. The difference is that our Changes Model is created automatically through manipulating the Screen Model in the GSME and does not have to be created manually.

5.5 Customization for Multiple Devices

The transformation of the same Discourse-based Communication Model to tailored GUIs for multiple devices typically requires similar customizations on all devices to achieve a satisfactory level of usability for, and consistency between all GUIs. We present our approach for GUI customization for a specific device, which assumes that a stable interaction design (i.e., Discourse-based Communication Model) and a Back-end (Stub) Source Code are already available. This assumption is fulfilled after a stable interaction design has been developed. For example, the *Create Interaction Design* activity of our tool-supported process for iterative and incremental interaction design and transformation rule-based GUI customization (shown in Figure 5.5) results in a device-specific set of `Transformation Rules`. The subsequent *Customize Device-specific GUI Model* activity results in a device-specific `Changes Model` and a `Style Sheet` that store the customizations performed with the GSME. These three artifacts are available after the GUI for the first device has been developed and they are potentially reusable for generating the GUI for all sub-sequent devices, together with the Communication Model.

Figure 5.14 shows our iterative process for GUI customization. In particular, this process details the *Customize Device-specific GUI* activity of our interaction development processes presented in Figures 5.1, 5.4 and 5.5 above, and requires a `Communication Model`, the corresponding `Back-end (Stub) Source Code` and an `Application-tailored Device Specification` of the device to render for as input. This process specifies how a GUI for a specific device can be customized. For customizing GUIs for multiple, i.e., n, devices, it has to be applied n times.

Let us assume here that we applied our iterative and incremental development process with customization through transformation rules shown in Figure 5.5 and that device-specific `Transformation Rules`, `Style Sheets` and a `Changes Model` are available from the development of a GUI for a previous device. All orange/dark activities in Figure 5.14 result in new, or customize existing artifacts. All yellow/light activities are evaluation activities.

The *Generate Screen Model Automatically* activity transforms the `Communication Model` to the `Screen Model`, under consideration of the `Application-tailored Device Specification`, fully automatically. The GUI designer can configure this transformation in three different ways. First, she can select one of the four pre-defined tailoring strategies presented above. Second,

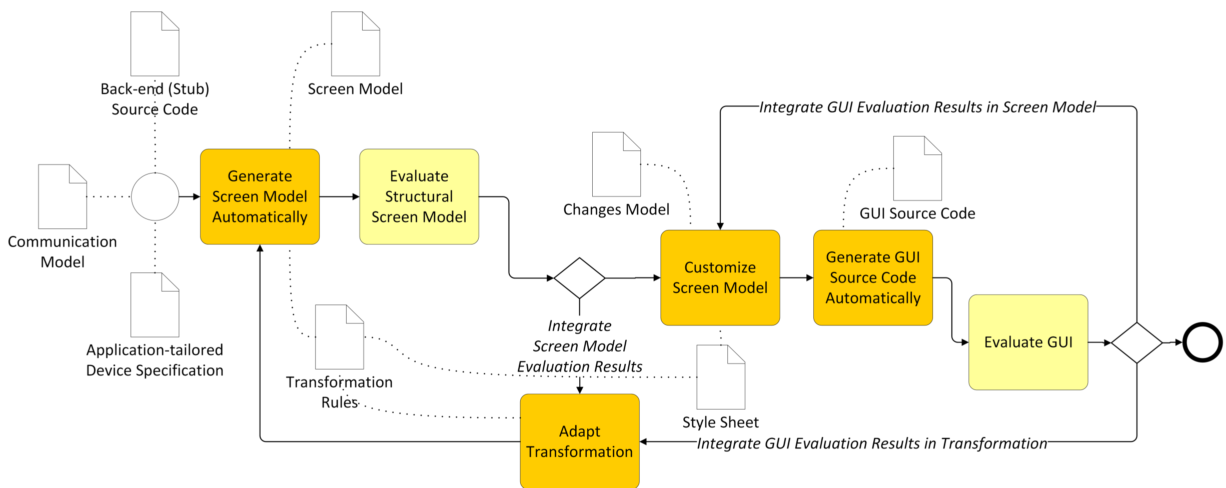


Figure 5.14: Iterative Process for GUI Customization

she can additionally provide the transformation rule set created during the customization of the previous GUI and third, she can provide the style sheet from the previous customization.

The `Screen Model` resulting from this activity can be evaluated in the subsequent *Evaluate Structural Screen Model* activity. The results of this evaluation can already lead to adaptations of the specific `Transformation Rules`, following the micro-iteration path labeled *Integrate Screen Model Evaluation Results*. Alternatively, the Screen Model can be customized in the *Customize Screen Model* activity. In case that a `Changes Model` from previous customizations is already available, these can be included and new customizations can be performed. These new customizations are, again, stored either in the `Changes Model` or in the corresponding `Style Sheet`.

The subsequent *Generate GUI Source Code Automatically* activity transforms the `Screen Model` to the `GUI Source Code`. Finally, the *Evaluate GUI* evaluates the running application, using the GUI under development and the `Back-end (Stub) Source Code`. We put stub in parentheses, because it does not matter for evaluating the "look & feel" of the GUI whether the back-end is already fully implemented or not. The results of this evaluation can either lead to further customizations of the Screen Model, represented through a second micro-iteration labeled "Integrate GUI Evaluation Results in Screen Model", or to adaptations of the transformation rules, represented through the macro-iteration labeled "Integrate GUI Evaluation Results in Transformation". The end-state of our customization process in Figure 5.14 is reached, if no further customizations are required.

Customizations of the PUs and the navigation between them through customization of the Communication Model are out of scope of our iterative GUI customization process for two reasons. First, changes to the Communication Model require most likely adaptations in the back-end, because the Communication Model specifies the functional interface between the back-end and the GUI. Second, changes in the Communication Model impact all GUIs that have been derived from this model.

In general, Communication Models are suited to specify all possible use cases [KPR12]. We did not include this step in our process, as the mapping from use cases to Communication Model has to be completed manually and is not supported by transformations. Nevertheless, if possible, the final application should also be tested against a given requirements specification.

Moreover, we refrained from defining roles through swim lanes, because most of the activities may involve more than one role. For example, it allows for faster iteration cycles if the interaction designer is involved in generating/customizing the GUI and creating a small application logic that allows for rapid prototyping. A clear assignment between roles and activities is helpful if decoupling them is desirable [EPV09], whereas a tight integration of the different activities allows for faster iterations. The balance between the number of roles and the time required for each iteration is typically determined by the size and complexity of the application to build and has to be assessed for each application individually according to our experience.

6 Evaluation and Results

This chapter presents the evaluation results of the conceptual approach presented in this doctoral dissertation. We start with presenting the evaluation of our iterative interaction development process, using a trial application for Bike Rental. Subsequently, we evaluate our iterative and incremental development process together with GUI customization for multiple devices, using a more complex application for Vacation Planning that is based on a commercial Web-site. We formulated the four best practices presented in Section 5.1 based on our experience gained when developing previous applications and applied them during the development of the Bike Rental and the Vacation Planning applications. Then we present usability evaluation results of generated GUIs for three different applications. First, we present the results of *Heuristic Evaluations* of two fully automatically generated GUIs (one for a desktop PC and one for a smartphone). Second, we present empirical data on the usability of our different tailoring strategies, based on *user studies* using three different GUIs for a simple Flight Booking application. Third, we present the results of a user study on the usability of a desktop and a smartphone GUI generated for the Vacation Planning application and relate them to the results of the evaluation of the corresponding commercial Web-site. At the end of this chapter, we summarize the results and key contributions of this doctoral dissertation and elaborate on how they support the hypothesis.

6.1 Evaluation of our Iterative Development Process using Bike Rental

We used a trial application for Bike Rental for the evaluation of our iterative process. We use this trial application to illustrate all activities of our process and present how we developed the corresponding interaction design in one micro- and a subsequent macro-iteration, using the Unified Communication Platform. We aimed for generating a desktop and a smartphone GUI. Following our best practice "BP2 – Assume a potentially infinite screen", we started with generating the GUI for the larger device, using the device specification for a Desktop Device shown in Table 6.1 and the "Screen-based Device Tailoring without Scrolling" strategy. The evaluation of our iterative development process is based on [RKP+14].

Our Bike Rental application supports three use cases: register a new user, rent a bike and return a bike. The rent and return bike use cases require the user to login first. An additional constraint was that a logged-in user should always have the possibility to cancel a rental or return process through logging out. Below we use an excerpt of the rent bike use case to illustrate the enactment of our new process.

Table 6.1: Desktop Device Specification

name	Desktop
resolution_x	1920
resoltuion_y	1200
dpi	72
defaultCSS	desktop.css
pointingGranularity	fine
toolkits	HTML
scrollWidth	1
scrollHeight	1

6.1.1 Creation of Initial Version and its Evaluation

Figure 6.1 shows an excerpt of the Discourse Model that is part of the initial version of the Communication Model. It captures the communicative interaction between a User and a System (depicted as interacting agents in the upper left corner). It is a key result of this activity, and we explain it here to illustrate what kind of modeling is to be done in its course.

The topmost IfUntil is assigned to the System (indicated through its yellow/light fill-color). It is a simplified version of this Procedural Construct modeling a simple loop, as it only has a so-called Tree branch but no Then branch and Else branch. Since this Tree branch does not have any Condition assigned, it models more precisely an endless loop. We use it in the Discourse Model to restart the System (i.e., to display the login screen again) after the interaction has been finished.

The IfUntil relation below has a Tree and a Then branch. This IfUntil is again assigned to the System, so that the condition of the Then branch `userSelectionReceived` has to be evaluated by the System. This particular condition specifies that the System needs to check whether the user either accepted the Offer to register (modeled through the Offer-Accept Adjacency Pair) or answered the OpenQuestion to login (modeled through the OpenQuestion-Answer Adjacency Pair) correctly. Both Adjacency Pairs (ADJs) can be seen in the upper part of Figure 6.1, depicted as diamonds with the corresponding Communicative Acts shown as rounded rectangles. The assignment of the Communicative Acts to either the User or the System is, again, indicated through their fill-color. In case that the System did not receive a user selection or the login was unsuccessful, the Tree branch is executed and the user may alternatively choose to register or to login. In case of an unsuccessful login attempt the Then branch of the Condition that carries the `loginFailed==true` condition is true and the user is informed that either username or password were wrong, in addition to having again the choice between registration and entering the login data again.

This elaborate part of the Discourse Model specifies a usual part of interaction recurring in many applications. So, it does not have to be created from scratch each time anew. In effect, this means an opportunity for reuse in the course of this first activity.

In case of a successful login, the discourse part depicted in the lower right part of Figure 6.1 is executed. Our initial interaction design did not contain the IfUntil relation shown in the rounded rectangle with a dashed outline, however. There was only the sequence relation depicted below this IfUntil, which specifies that the user needs to select a bike in the first step (modeled in the Nucleus branch labeled with "1") and either confirm (i.e., accept) or reject this selection in the

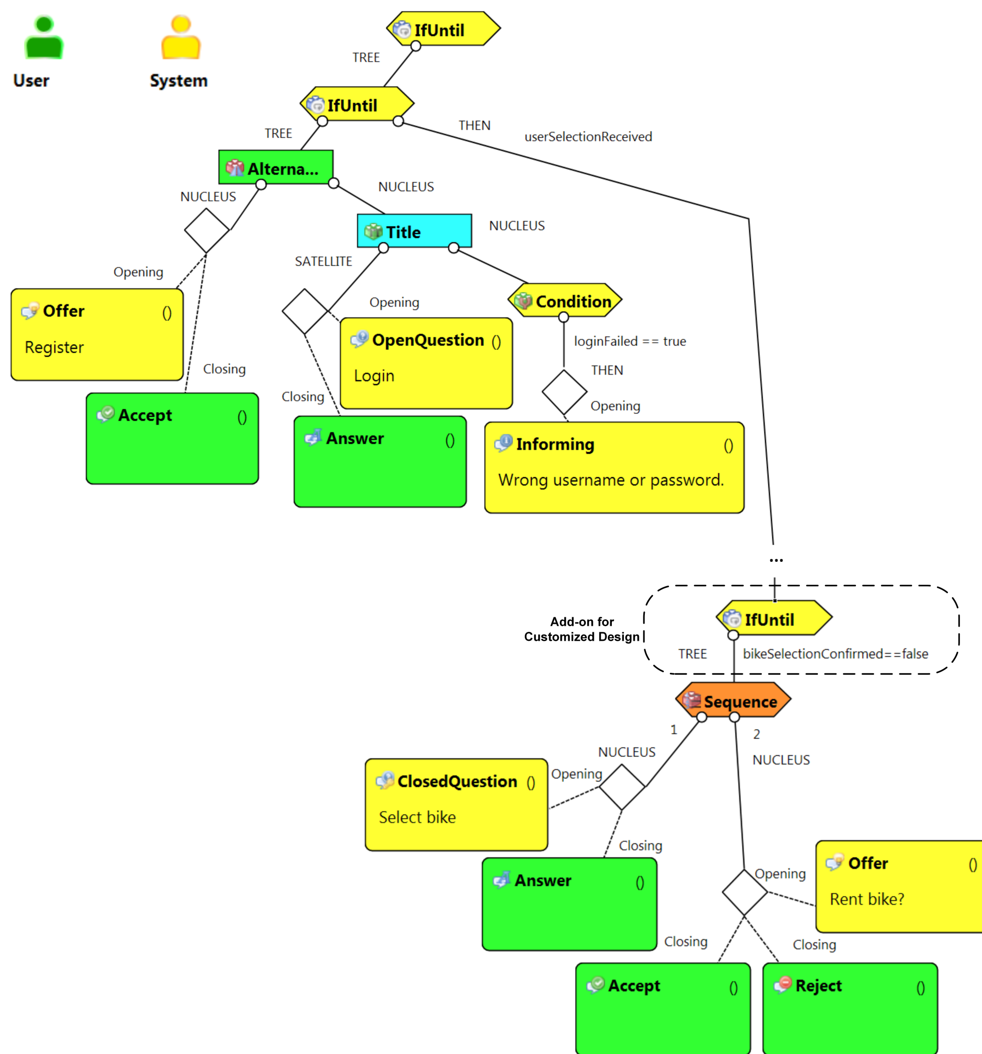


Figure 6.1: BikeRental Discourse Model Excerpt [RKP+14]

second step (modeled through the Offer-Accept/Reject Adjacency Pair in the Nucleus branch labeled with "2").

This model part framed in Figure 6.1 passed an informal evaluation by two (experienced) interaction designers when evaluating it against the use case specification at the end of the activity *Create Initial Communication Model* of our iterative development process. The complete Discourse-based Communication Model created as an initial interaction design based on the three use cases was fully-automatically transformed into a Screen Model and the corresponding GUI source code using the tools provided by UCP. This allowed for an informal evaluation of the screens specified in the Structural Screen Model using the corresponding graphical editor. Figure 6.2 shows the visualization of the automatically generated screen of the Structural Screen Model for the confirmation of a previously selected bike. The "Ok" button confirms the selection by sending the Accept Communicative Act depicted on the bottom of Figure 6.1 and the "Cancel" button rejects the selection by sending the Reject Communicative Act. The examination of this screen revealed that the logout functionality was not available, indicating that its integration had not been considered adequately in the initial interaction design.

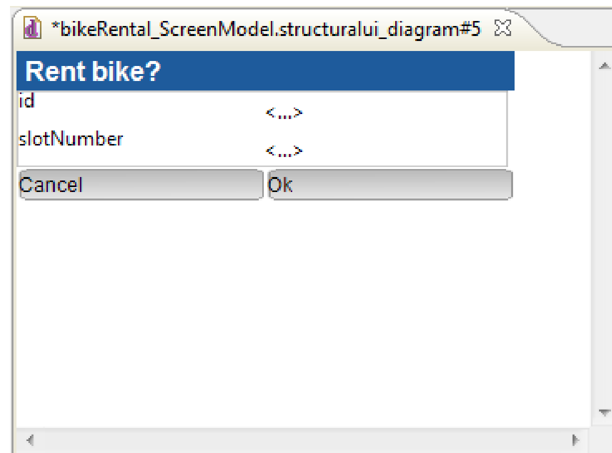


Figure 6.2: Visualized Screen of the Structural Screen Model [RKP+14]

6.1.2 Enactment of Micro-iteration

The interaction designers immediately corrected this flaw in the activity *Customize Communication Model*, in effect performing a micro-iteration following the path labeled *Integrate Screen Model Evaluation Results* in our process. This entailed the automatic generation of the GUI from the customized interaction model. The subsequent evaluation of the resulting Structural Screen Model did not reveal any flaws.

So, the process enactment proceeded to the activity *Develop Back-end Stub*, where the stubs in the automatically generated functional interface have been implemented. Their availability allowed for running the resulting prototype. Figure 6.3 shows a screen shot of the screen for the confirmation of a previously selected bike. Much as the corresponding screen in the newly generated Structural Screen Model, it contains a “Logout” button.

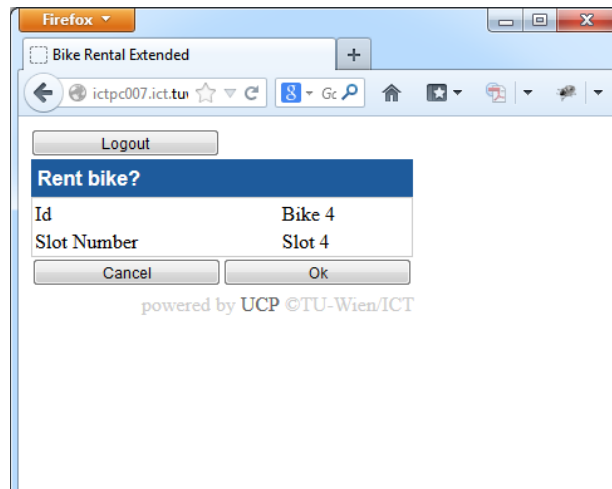


Figure 6.3: Confirm Selection and Rent Bike Screen of the Final GUI [RKP+14]

More importantly than showing real screens of the GUI, this running prototype was available for directly evaluating the interaction. In fact, four usability experts (different from the two experts

that evaluated the screen model) performed a Heuristic Evaluation of this first running prototype. It revealed several flaws regarding its external behavior.

One flaw, detected as confusing behavior in the course of the Heuristic Evaluation, was that the user became logged out in case she rejected the confirmation of a selected bike. Figure 6.4 illustrates this behavior in the form of a simple state-machine. The transition between the "Bike Selection" and the "Confirm Selection and Rent Bike" screens is triggered by selecting a bike (i.e., sending the Answer Communicative Act to the ClosedQuestion depicted in the lower right part of Figure 6.1). The subsequent confirmation (i.e., sending the Accept Communicative Act) leads to the "Bike Selection Confirmed" screen, while Reject leads to the "Bike Selection NOT confirmed" screen. The latter was, in fact, the login screen, as the Reject Communicative Act terminated the Then branch of the second topmost IfUntil relation in Figure 6.1, leading to the execution of its Tree branch through the endless loop of the topmost IfUntil.

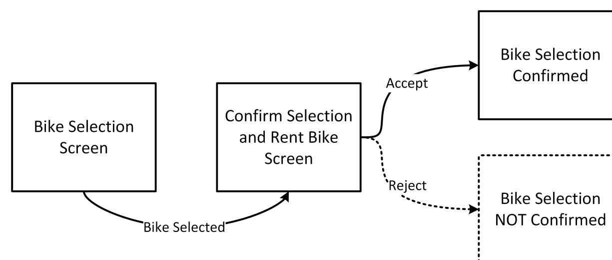


Figure 6.4: Initial Interaction Design Behavior Excerpt [RKP+14]

6.1.3 Enactment of Macro-iteration

Correcting this and the other flaws in the interaction design detected through the Heuristic Evaluation (for details see Subsection 6.3.1 and [RWP+13]) led to a macro-iteration, following the *Integrate Interaction Evaluation Results* path in our process. In particular, an interaction designer added another loop into the Discourse Model of the interaction design to correct the confusing behavior described above as suggested by the usability experts. This loop is modeled through the IfUntil shown in the rounded rectangle with the dashed outline in Figure 6.1. The condition of this IfUntil's Tree branch specifies that it is executed as long as the bike selection has not been confirmed. This leads to the behavior for this part of the model as specified in the state-machine in Figure 6.5. The difference to the behavior shown in Figure 6.4 is that the Reject transition now leads back to the Bike Selection screen.

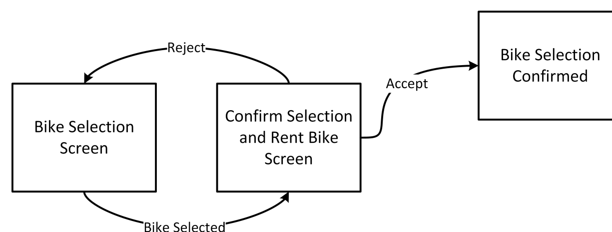


Figure 6.5: Customized Interaction Design Behavior Excerpt [RKP+14]

After finishing the customization of the Communication Model, another version of the GUI was generated automatically. The evaluation of the interaction of its corresponding running prototype

did not reveal any further flaws in the external behavior when verified against the results of the previously performed Heuristic Evaluation. This indicated the achievement of a stable interaction design in the form of a Communication Model, which allows customizing the GUI for different devices and implementing a more elaborate back-end concurrently, as specified in the two activities shown at the bottom of Figure 5.1. These activities are, however, out of scope for this section.

It is interesting to observe that this is an *iterative* process for interaction design, while software development in general is usually iterative *and incremental*. So, a more or less complete initial interaction design is to be created. This works out fine for small (and possibly medium-sized) applications, but it may be insufficient for large ones. Our incremental process with GUI customization through transformation rules offers additional flexibility for the designer as it allows for changing / extending the Communication Model and customizing the GUI iteratively.

An important property of GUI prototypes is their fidelity. Creating a low-fidelity prototype is obviously cheaper and faster than programming a high-fidelity one. The latter is, however, better suited for evaluations. In our process, UCP creates both a medium-fidelity and a more or less perfect-fidelity prototype of the GUI. For running the latter, however, at least a stub of the back-end is required. Still, the time and effort for producing these kinds of prototype is small compared to traditional approaches, especially for iterating on them in the course of developing alternatives.

6.2 Evaluation of our Iterative and Incremental Development Process with Customization for Multiple Devices using Vacation Planning

We built a Vacation Planning application according to our process for iterative and incremental interaction design and transformation rule-based GUI Customization (see Figure 5.5), in effect evaluating this process. In particular, we use this trial application in this section to illustrate all activities of our process and present how we developed the corresponding interaction design incrementally in several micro- and macro-iteration, using UCP tool support. For generating the desktop GUI, we configured the UCP UI generation framework (UCP:UI) to use the device specification for a Desktop PC shown in Table 6.1 and the "Screen-based Device Tailoring without Scrolling" strategy. This evaluation of our iterative and incremental development process is partly based on [RPK+14].

Our Vacation Planning application is based on a commercial accommodation booking Web-site of an Austrian province and implements a subset of this Web-site's functionality. In particular, our application supports searching for an accommodation either through text search or through more specific search masks. In addition, it provides information on events, articles that report on different topics, and on how to get to this Austrian province. Finally, it allows a user to send a booking request to a specific accommodation or to book an accommodation directly. Below we use excerpts of the corresponding Communication Model to illustrate the enactment of our new process.

6.2.1 Initial Iteration

The basis for our interaction development was a set of tasks that should be supported through the Vacation Planning application to build, and an existing commercial Web-site that supported

them. Such tasks were, for example, to get information on events, to book a specific accommodation, or to find out how to get there. These tasks were already supported by the commercial Web-site, and we basically re-engineered its interaction. Our aim was to allow for a comparative usability evaluation of our application through a user study in the end (which is summarized in Subsection 6.3.3).

On the commercial Web-site the interaction required to support the tasks was triggered through links labeled *Home*, *Search Accomodation*, *Plan Vacation*, *Get There* and *Events*. The Web-site allowed for switching between them at any time, which decouples the interaction attached to each category and facilitates incremental development of the interaction model.

The *Home* link, for example, led to the start-page, which displayed a welcome message and presented a list of events and a list of articles. Selecting a specific event or article led to a Web-page with more detailed information on the selected event / article.

When developing our interaction design, we started with modeling these alternative selections and the interaction provided by the start-page. Figure 6.6 shows an excerpt of the corresponding Discourse Model. This model has German content descriptions for the Communicative Acts, because the language of the application was German and these descriptions are used as default labels by our GUI generation framework. We will use English translations in this section, providing the original German words in italics next to the translation.

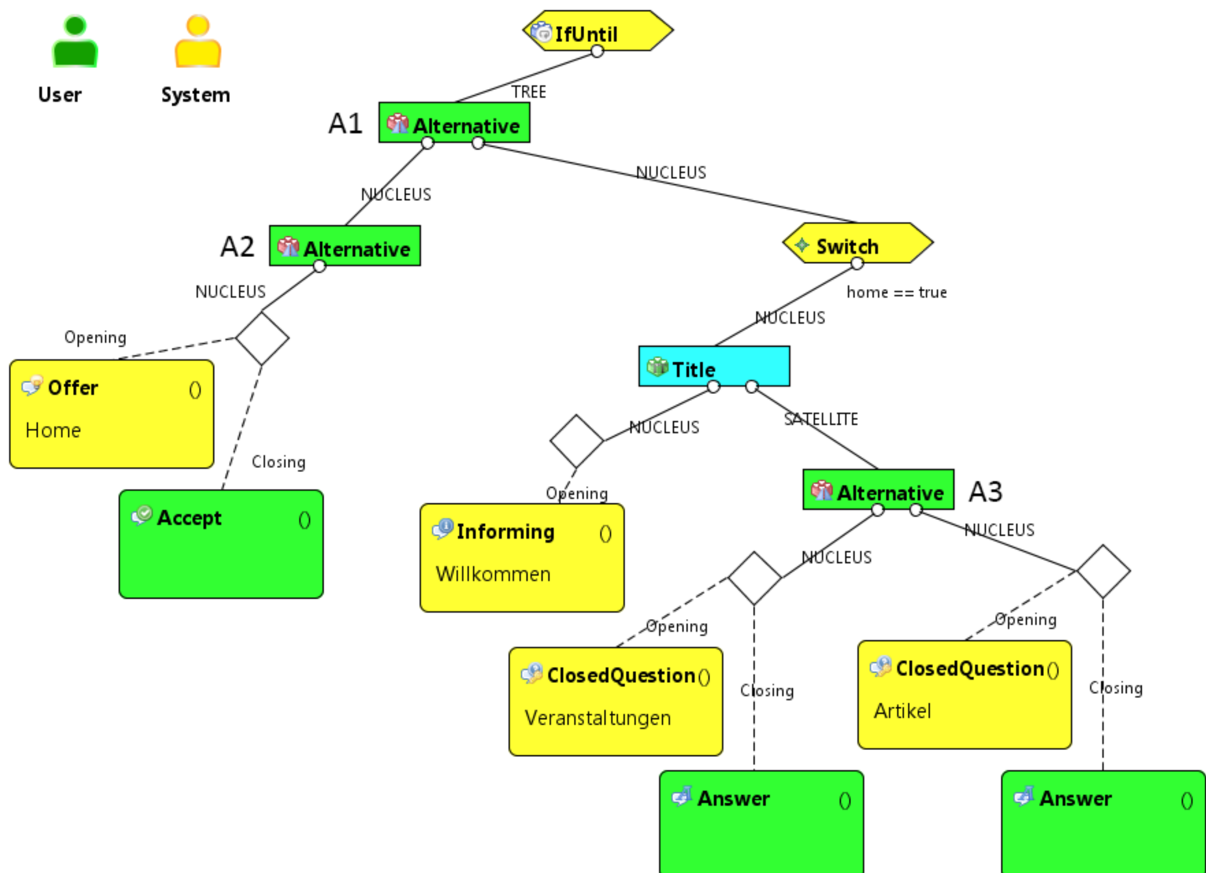


Figure 6.6: Excerpt of Initial Vacation Planning Discourse Model [RPK+14]

The interacting parties in our Communication Model are a User (green / dark fill-color) and the System (yellow / light fill-color), depicted in the upper left corner of Figure 6.6. The top-level `IfUntil` relation is assigned to the System (indicated through its yellow / light fill-color) and specifies a so-called `Tree` branch without a condition and no `Then` or `Else` branches. This models an endless loop used for restarting the application (i.e., to display the home screen again) after the interaction has been finished.

The Alternative relation labeled A1 has been assigned to the User, which means that the User can alternatively perform any interaction specified in its `Nucleus` branches. Its left Nucleus contains another Alternative relation (A2), which we used to model the selection between the five links that should be available at any time for the user during the run-time of the application. We built our interaction model incrementally starting with the Home link, which is modeled through the `Offer-Accept` Adjacency Pair. The corresponding interaction is modeled in the second Nucleus branch of Alternative A1. This nucleus contains a `Switch` relation that has been assigned to the System. This means that the condition assigned to the Switch's Nucleus branch (i.e., `home==true`) is evaluated by the System. This condition is true when the application is started and can be set to true at any time by the User through accepting the Home Offer (modeled through the corresponding Adjacency Pair as described above).

The interaction of the start-page has been modeled in the Nucleus branch of the Switch relation with the condition `home==true`. It contains a `Title` relation, whose Nucleus contains the `Informing` Communicative Act presenting the welcome message, and whose Satellite contains another Alternative relation (A3). This Alternative relation links two `ClosedQuestion-Answer` Adjacency Pairs, one for the events (*Veranstaltungen*) and one for the articles (*Artikel*).

In addition to the Discourse Model, we also created the initial Domain-of-Discourse Model and the Action-Notification Model in an iterative way. The initial Domain-of-Discourse Model is shown in Figure 6.7. It defines the classes `StartObject`, `Event` (*Veranstaltung*) and `Article` (*Artikel*) with their respective attributes. Moreover, it defines `EventType` (*Veranstaltungstyp*) enumeration, which is used by the `type` (*typ*) attribute of the Event class.

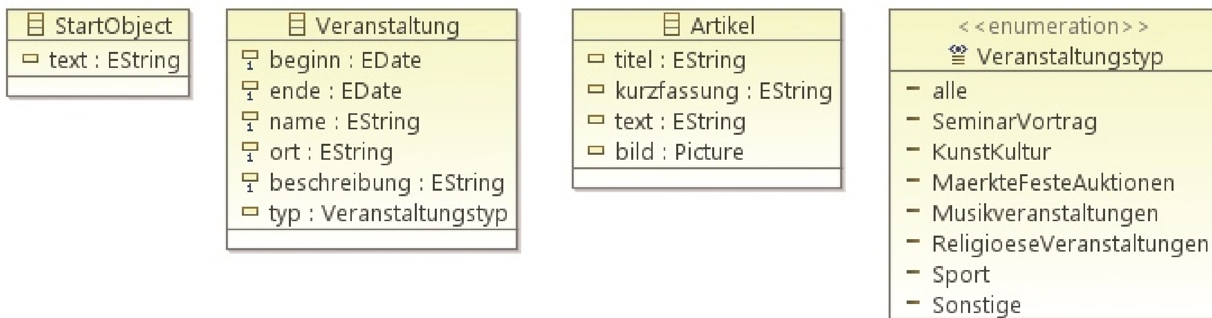


Figure 6.7: Initial Vacation Planning DoD Model [RPK+14]

The initial ANM contained a `home` Action only, as the remaining actions used in our initial Communication Model were already specified in the *basic ANM*. Both, the concepts specified in the DoD Model and in the ANM were referenced through the Propositional Content of the Communicative Acts in our Discourse Model.

After having created these three models, we generated the corresponding GUI and evaluated the Screen Model. The Screen Model did not fit our expectations for three reasons. First, the *basic* transformation rule that matches a ClosedQuestion-Answer Adjacency Pair with the content of

an EObject renders two widgets (a label with the attribute name and a widget for the attribute value) for each attribute of the EObject. This rule was executed (fired) for the Article and the Event ClosedQuestion-Answer Adjacency Pairs (ADJs) and created widgets for all Event and Article attributes defined in the DoD Model. Instead, we only wanted to render a subset of attributes. Second, this transformation rule creates a radio button to allow for the selection of a specific Object instance and a button to submit this selection. Instead, we wanted to provide a button for each entry that directly submits the selection. Third, the automatically generated layout did not match the layout that we wanted to achieve. So, we performed the micro-iteration labeled *Integrate Screen Model Evaluation Results* in our iterative and incremental interaction design and GUI customization process shown in Figure 5.5, and created *custom* transformation rules for these two ClosedQuestion-Answer ADJs in the *Adapt Transformation* activity.

Figure 6.8 shows the custom transformation rule that we created for the Article ClosedQuestion-Answer ADJ. The LHS matches a `ClosedQuestion-Answer ADJ` with the content `select one from many Artikel`, which specifies that the corresponding interaction party needs to provide a list of Articles (*Artikel*) from which one shall be selectable. The RHS specifies a `Panel` with the name Artikel_Container and the `Style` thinBorder. This style does not have to be defined explicitly, as it is already defined in the style sheet of the *basic* transformation rule set. Alternatively new styles can be defined in an additional style sheet. This panel contains a Heading_Artikel `Label` and a `List Panel` which contains a Panel with the name Artikel, again with the `Style` thinBorder, which contains the Articles. This Panel specifies how each list entry is to be rendered through its children. The Panel in Figure 6.8 specifies that the Title (*title*) attribute of an Article is rendered as a `Button` and the Short Description (*kurzfassung*) attribute as a Label. The text (*text*) and picture (*bild*) attribute are not rendered. The layout of all containers and their children is explicitly specified through the layout data attached to the widgets in the RHS of the transformation rule.

6.2.2 Incremental Extensions

In the subsequent iteration we modeled the interaction following the selection of a specific event or article on the start-page modeled in our first incremental iteration. This interaction intuitively belongs to the home Sub-Discourse presented above, but we modeled it in a new branch of the Switch relation instead, because it should be reachable from different points of the discourse (see Figure 6.9).

This Switch branch specifies the condition `gewaehlterArtikel!=null`, and contains an `OrderedJoint` Relation that links an Offer-Accept ADJ, which models the back functionality, and the Informing ADJ that models the presentation of the selected Article. After extending the Communication Model, we automatically generated the Screen Model anew, employing our two custom transformation rules for the Event and Article ADJs. We evaluated the corresponding Screen Model and subsequently started to develop the back-end stub to evaluate the interaction.

Figure 6.10 shows the final GUI for the Home branch of the Switch relation (as sketched in Figure 6.6), rendered for the desktop device employing the custom transformation rules for the Event and the Article ClosedQuestion-Answer ADJs.

After having implemented the Back-end Stub, we tested the interaction, which was already valid. However, the GUI still required further customization. So, we added two more custom rules (shown in Figure 6.11), following the *Integrate Application Test Results* path in Figure 5.5. In

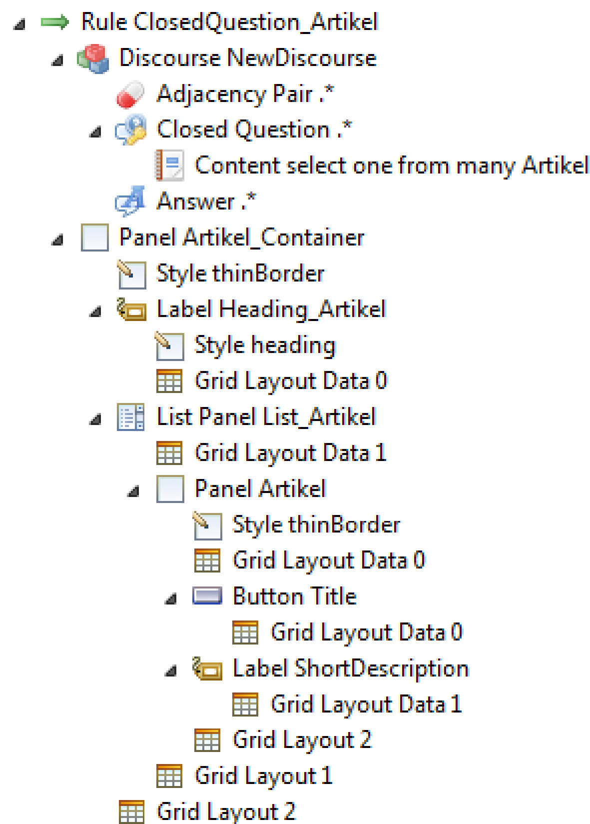


Figure 6.8: Custom Rule for ClosedQuestion-Answer ADJ with Content Article (*Artikel*)

particular, we wanted to render only a selected set of Article (*Artikel*) attributes of the new Informing ADJ in Figure 6.9 and we wanted to achieve a specific layout. To achieve this, we created the custom rule shown in Figure 6.11(a), which renders an Adjacency Pair with an Informing Communicative Act and an Article (*Artikel*) as its content, creating the desired GUI part.

The second custom rule shown in Figure 6.11(b) was created for the Offer-Accept ADJ that models the back functionality (i.e., content `zuruecksetzen one EObject`). The Action `zuruecksetzen` with the Attribute `element` was added to the ANM and models a back functionality, which explicitly allows the user to reset the value of a specific element (i.e., variable). We used this Action to reset the selected article, which we stored in a specific variable that was defined in the Propositional Content of the ClosedQuestion-Answer ADJ. The RHS of this rule creates a Panel with a Button. We illustrate this rule here, because it uses the Layout Hints introduced in Section 4.3. The specific Layout Hint used here specifies `alignment-x` as left and `alignment-y` as top, positioning the “Zurueck” Panel in the upper-left corner of its parent container. We selected this configuration because the Back Button in Web-browsers is also typically found in the upper-left corner.

We used the same Communication Model pattern six times in the final Communication Model, so the same custom rule was fired for each pattern. Specifying the layout explicitly would have been more effort, as we would have needed to design six custom rules, which would match the Offer-Accept ADJ, all its siblings and its parent Relation. In fact, this rule illustrates the reusability of transformation rules.

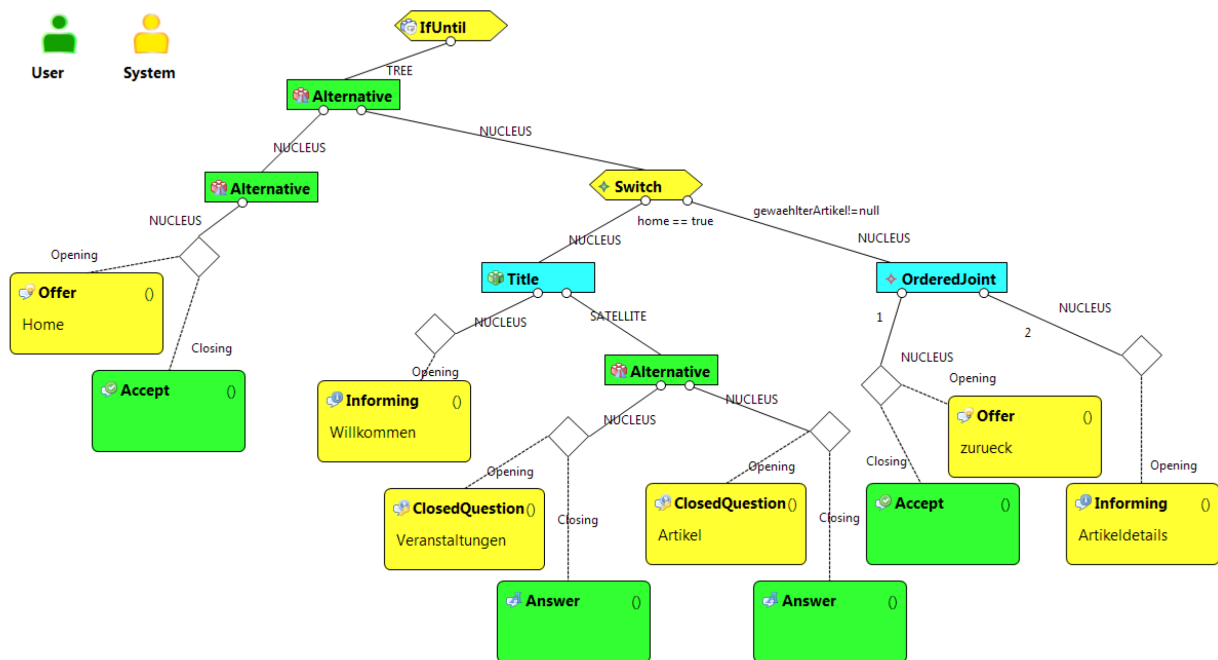


Figure 6.9: Initial Vacation Planning Discourse with Article Details Extension

After having completed these two transformation rules, we further extended the Communication Model through adding a Switch branch for the Event details, because this branch was again reachable from different points of the Communication Model, just like the Article details branch. In general, we modeled all interactions that should be reachable from different points in the interaction model as sub-branches of the Switch relation, extending the existing interaction model and developing the final interaction model in increments. More complex branches were refined iteratively, before adding another increment. Figure 6.12 shows the GUI rendered for the final Communication Model, employing the two custom rules presented in Figure 6.11.

The left side of Figure 6.12 shows the final navigation panel with the Home, Find Accommodation (*Unterkunft finden*), Plan Vacation (*Urlaub planen*), Arrival (*Ankunft*) and Events (*Veranstaltungen*) Buttons as well as a text field and a Search (*Suche*) Button for text search. To achieve the desired layout of the buttons, we designed a custom rule that matches the Navigation Alternative A2 of the final Discourse Version shown in Figure 6.14. This rule is shown in Figure 6.13 and explicitly specifies the layout in the rule's RHS. This *custom* transformation rule additionally specifies the names of the Communication Model elements to match, in contrast to the *basic* transformation rules presented above, which do not match a specific name (i.e., the name is specified as the regular expression ".*"). In this sense, the custom layout rule presented in Figure 6.13 is more specific than the custom rules presented in Figures 6.11 and 6.12 above, because it does not only match a more specific pattern of Communication Model elements (in comparison to the corresponding *basic* transformation rules), but also specific names of these elements.

The layout rule in Figure 6.13 uses `Mappings` to assign each link of the Rules Communication Model pattern (i.e., LHS) to a specific element (i.e., `Panel`) of the rule's RHS. The Communication Model pattern specifies the Navigation Alternative (A2 in Figure 6.14) with the name "NavigationUndBuchung". The direct children of a relation are `Link` elements, which link Discourse nodes. In our layout rule, we specified six Link elements that each link the Alternative

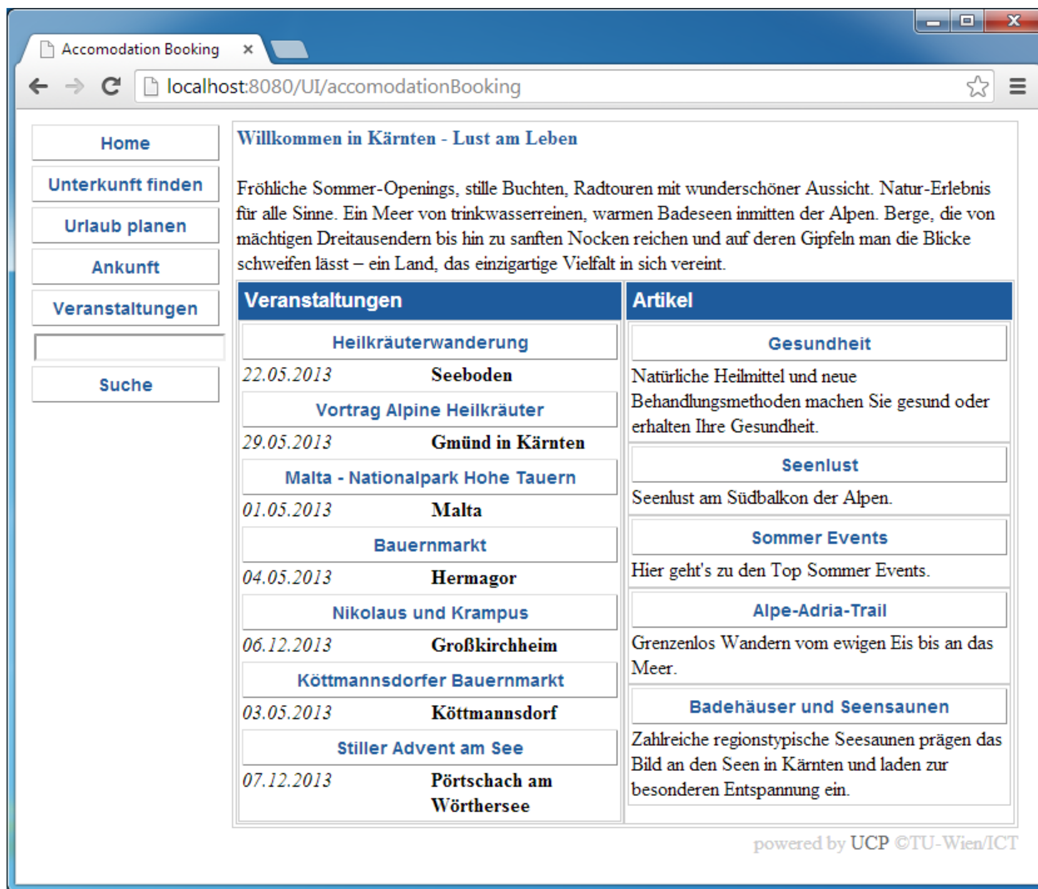


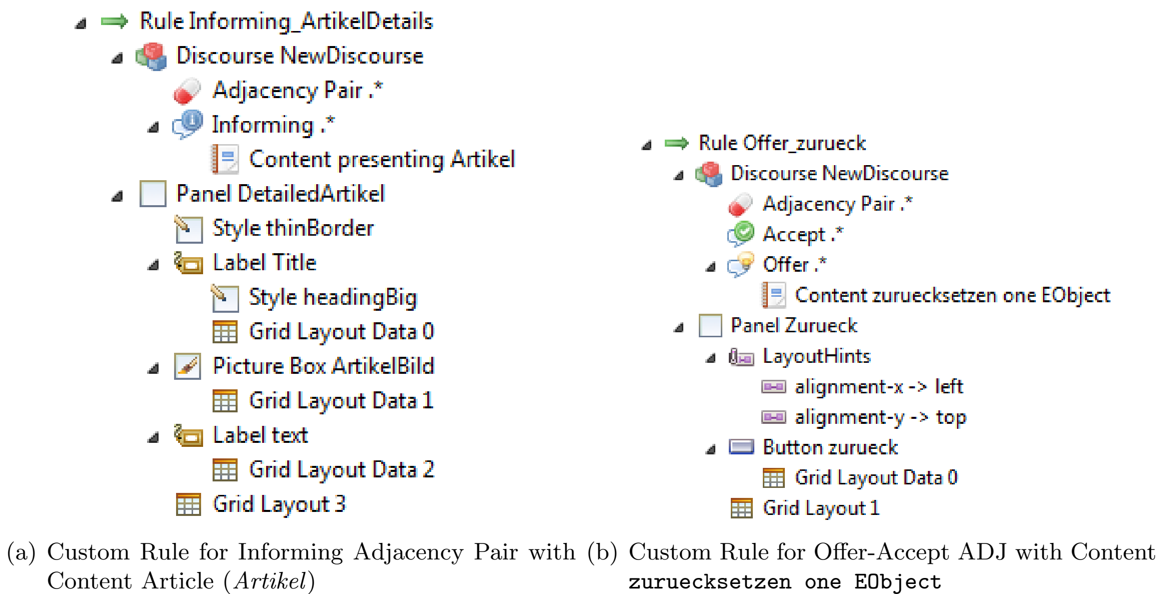
Figure 6.10: Home Screen Customized through Transformation Rules

Relation to one of its six child nodes. These child nodes are modeled through the six Adjacency Pairs ("Home", "FindUnterkunft", "PlanVacation", "HowToArrive", "StartSearch" and "ShowVeranstaltungs"). This pattern exactly matches the navigation part of the final Vacation Planning Communication Model. The layout for the generated Structural UI part is explicitly specified through the `Grid Layout Data` elements in the `Panel` structure of the rule's RHS.

Figure 6.14 illustrates the structure of the final Discourse Model. As already visible in the Screenshot in Figure 6.12, this model offers six options in the navigation (i.e., Alternative) branch of Alternative relation A1. Five of these options were Offer-Accept Adjacency Pairs, which are sketched in the left side of Figure 6.14 through the Home and the Events (*Veranstaltungen*) Adjacency Pairs with dots in between. The sixth branch contains the text search (*VeranstaltungenSuche*), modeled as OpenQuestion-Answer Adjacency Pair.

Each Offer-Accept Adjacency Pair triggers the corresponding interaction, modeled as a branch of the Switch relation. The Switch branch of Alternative A1 finally contains eleven branches due to the additional branches for interaction that was reachable from different points of the final discourse, which are sketched on the right side of Figure 6.14.

The five branches corresponding to the Offer-Accept Adjacency Pairs are sketched through the Home and the Events (*Veranstaltungen*) notes with dots in between, on the right side of Figure 6.14. The remaining six branches, which model interaction reachable from different points in the discourse, are sketched through the dots between the Events (*Veranstaltungen*) and the



(a) Custom Rule for Informing Adjacency Pair with Content Article (*Artikel*)

(b) Custom Rule for Offer-Accept ADJ with Content `zuruecksetzen one EObject`

Figure 6.11: Custom Rules for Article Details Screen

Payment (*Bezahlung*) notes. The first of these six branches, specifies the display of text search results, which typically contains a list of accommodations, events and articles. Each item in these three lists can be selected to provide further details, which triggers the interaction in the corresponding branch of the Switch relation. We provided three branches for this interaction, one for Accommodation details, which also allowed for booking a specific accommodation, one for Event and one for Article details. The remaining two branches model the interaction for getting information on how to get there by a specific means of transport (i.e., plane, train or car) and the interaction required for completing the booking process for a specific accommodation through payment.

Overall, we performed 11 incremental iterations while developing the Communication Model. After more complex increments, we evaluated and refined the Communication Model iteratively. Extending and refining the Discourse Model also included extensions and modifications of the DoD Model and the ANM. The final DoD Model specifies 18 classes and 8 enumerations, and the final ANM specifies 11 Actions and 2 Notifications.

The final Communication Model consists of 168 Discourse Elements, i.e., 84 Communicative Acts, 49 Adjacency Pairs and 35 Discourse Relations. This Communication Model corresponds to 86 Communication Model patterns that need to be matched by transformation rules to completely transform it into a GUI. The number of patterns is smaller than the number of Discourse Elements, because the atomic source model units are typically Adjacency Pairs where the opening Communicative Act is sent by the system, and Relations. In fact, 2 rules were fired for Communicative Acts, 49 rules for Adjacency Pairs and 35 rules for Discourse Relations.

35 of the 86 Communication Model patterns were matched by two transformation rules of the *basic* rule set, because this rule set already provides rendering alternatives (see Section 4.2). This means that using the *basic* rule set generates a search space with $2^{35} = 34.359.738.368$ GUIs that can, in theory, be generated for the given Communication Model. We call such GUIs that can be generated in theory "possible" GUIs in the remainder of this doctoral dissertation for the sake of brevity. A large number of possible GUIs typically requires more automated

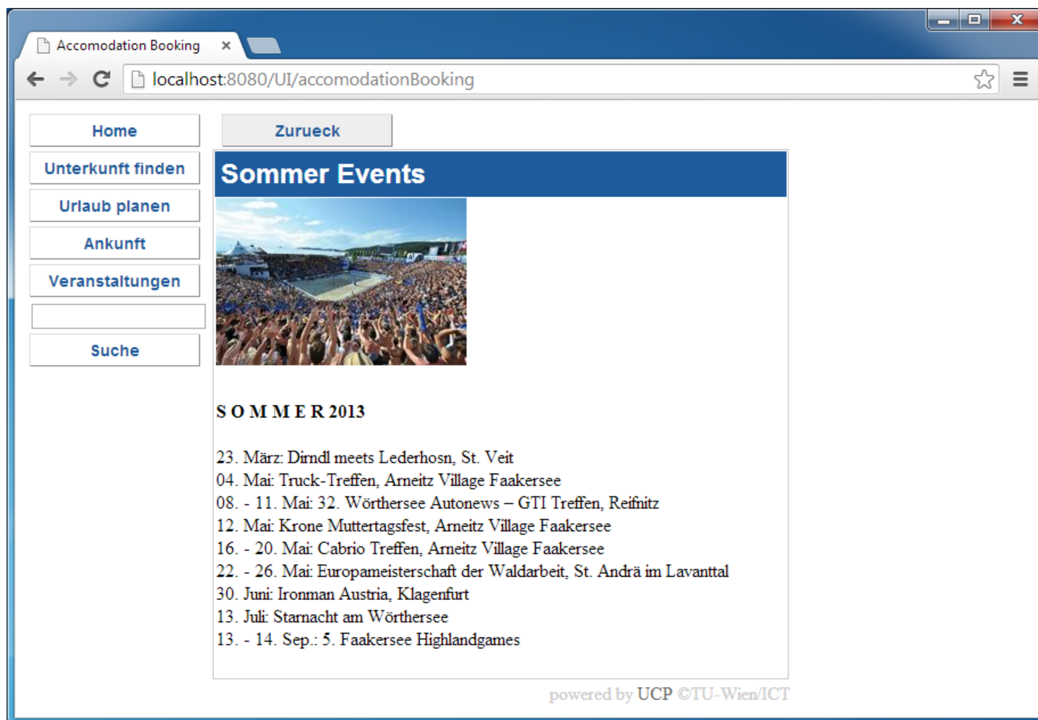


Figure 6.12: Article Details Screen Customized through Transformation Rules

tailoring loops (as presented in Figure 4.15 above). This leads to longer iteration cycles and an unpredictable GUI. To increase the predictability of the resulting GUI and to reduce the time required for generating an optimal GUI, we already started customizing it through transformation rules during the iterative and incremental development of the Communication Model, following the process defined in Section 5.3. We used transformation rules from the `generic-custom` rule set wherever feasible. Through using generic-custom transformation rules, we reduced the number of patterns that previously matched two rules to 8. This means that we reduced the number of possible GUIs to $2^8 = 256$. Additionally, we customized how specific Communication Model Patterns were rendered (i.e., which attributes, which layout and style) through new rendering rules like the ones presented in Figure 6.11, which are defined in the so-called `custom` rule set.

Table 6.2 presents an overview of the *basic* transformation rules fired for generating the Desktop GUI. SELF and OTHER in the rules' names (shown in the second column) refer to the role that a particular Relation or the Opening Communicative Act of an Adjacency Pair is assigned to. SELF is matched to the Communication Model role that has been selected in the launch configuration (i.e., the one representing the user) and allows our transformation engine to select the appropriate transformation rules. The third column in Table 6.2 shows how often a specific rule was fired when transforming the Vacation Planning Communication Model. For generating the Desktop GUI, 22 *basic* transformation rules were fired for 39 patterns.

Table 6.3 shows the three alternative transformation rules from the *basic* rule set that alternatively match 8 Communication Model Patterns, potentially allowing for generating $2^8 = 256$ GUIs. None of these rules was fired for the Desktop GUI, however, because the initially generated GUI already fit and no further tailoring loops were required.

The initially generated GUI already satisfied the Desktop constraints, because we manually reduced the number of alternatives through using (generic-)custom transformation rules. Table 6.4

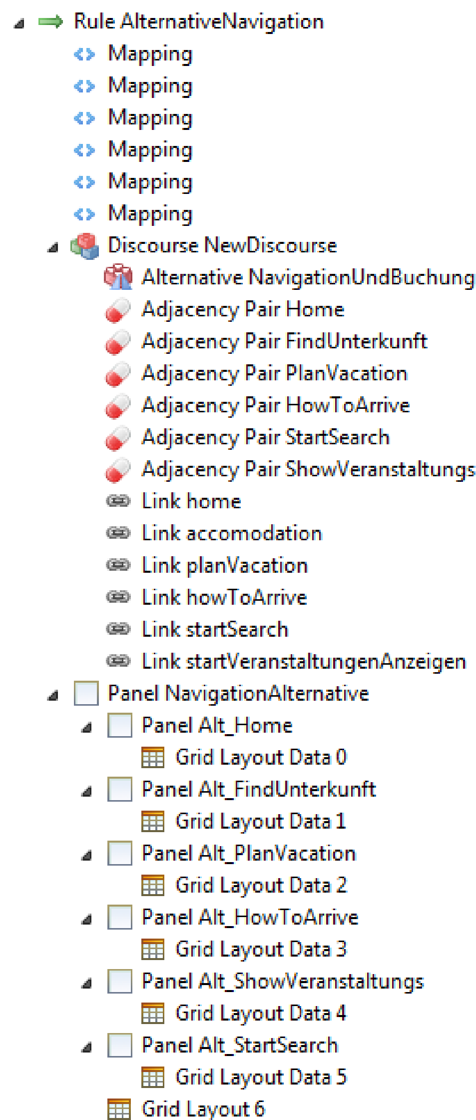


Figure 6.13: Custom Rule for Layout

shows that six transformation rules that were already available in the generic-custom rule set were fired for 15 Communication Model Patterns. The rules 3 to 6 match specific Relations that were also matched through a *basic* transformation rule. As all generic-custom rules inhibit the corresponding *basic* rules from firing, they reduced the number of alternatives on 13 Communication Model patterns. Transformation rules 1 and 2 provide alternative renderings for patterns that previously had no alternative, so these rules do not reduce the number of rendering possibilities, but are rather used to partially achieve a specific GUI.

Table 6.5 shows the 16 rules of our `Vacation Planning` transformation rule set that we specifically designed for the Vacation Planning application. Rules 1 to 3 were designed to achieve a specific layout, either explicitly or through Layout Hints. Rules 4 to 16 were used to render only specific attributes of a certain DoD concept and to achieve a specific layout and style of the resulting GUI. Rules 1, 2, and 12 to 16 furthermore reduce the number of possible GUIs, because they match Communication Model patterns that are also matched by an alternative transformation

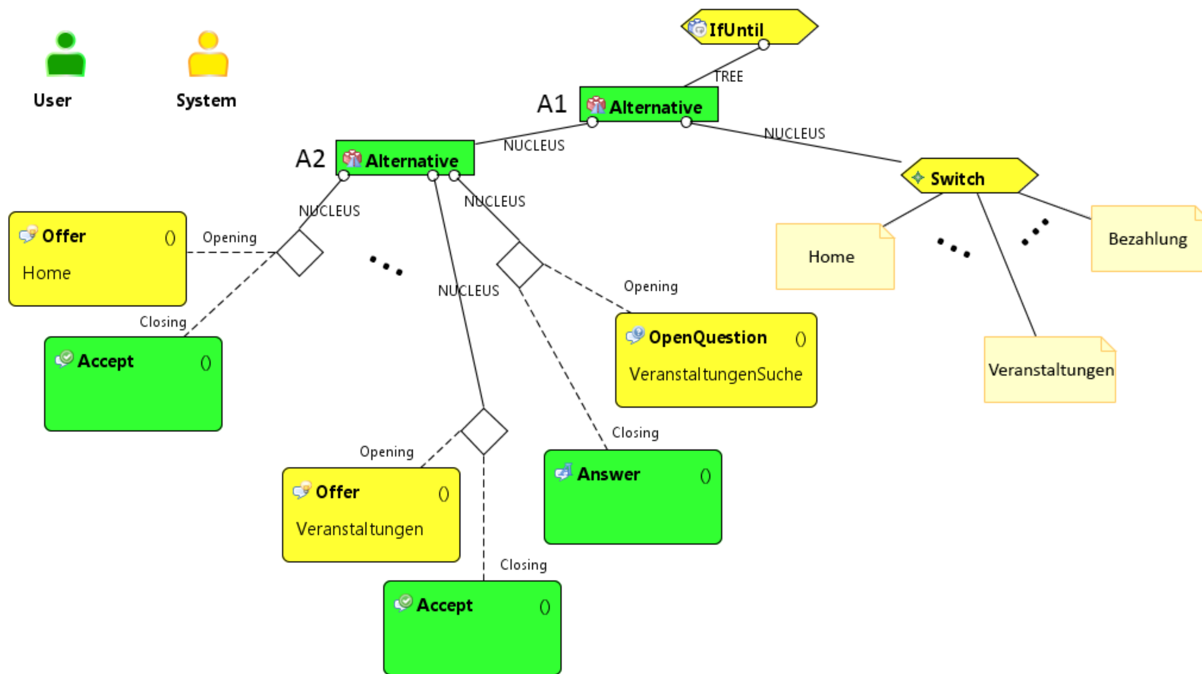


Figure 6.14: Excerpt of Final Vacation Planning Discourse Model [RPK+14]

rule. These 7 rules were fired fo 14 Communication Model patterns and decrease the number of possible GUIs.

The 16 custom rules were fired for a total of 32 Communication Model patterns, the 6 generic-custom rules were fired for 15 patterns and the 22 *basic* transformation rules were fired for 39 patterns. In total, 41 different transformation rules were used, but only 16 (i.e., the custom rule set) had to be created from scratch. These numbers show that customizations through transformation rules are typically applied several times and potentially decrease the effort in comparison to completely manual customization.

Through firing (generic-)custom transformation rules for patterns that potentially allowed for alternatives, we were able to decrease the number of GUIs that can be generated for the given Communication Model to

$$2^{35} * \frac{1}{(2^{13} * 2^{14})} = 2^8 = 256.$$

Through customizing the GUI, we also achieved that no automated tailoring loops were required when rendering for the Desktop device, even with our "Screen-based Device Tailoring Strategy without Scrolling", because the first generated GUI already satisfied our constraint (i.e., fit the available screen space). This customization led to a more predictable GUI through firing custom rules and allowed for faster iterations, as no tailoring loops were required for the Desktop GUI.

After achieving a stable Communication Model and Vacation Planning rule set we further customized the GUI through adding style rules in a specific style sheet, the so-called *Desktop CSS*. This corresponds to the *Customize Device-specific Screen Model* activity in our incremental and iterative development process (see Figure 5.5 above). In parallel we implemented the complete back-end, which corresponds to the *Develop Complete Back-end* activity in Figure 5.5 above.

Table 6.2: Basic Transformation Rules fired for Rendering the Vacation Planning Desktop GUI

Basic Communicative Act Rules		*Matches*
1.	Basic SELF Request Constant	1
2.	Basic OTHER Accept Constant	1
Basic Adjacency Pair Rules		
3.	Basic SELF AdjacencyPair	1
4.	Basic OTHER Offer-Accept One Object	1
5.	Basic OTHER Informing One Object	1
6.	Basic OTHER OpenQuestion-Answer One Object	2
7.	Basic OTHER ClosedQuestion-Answer Many EnumLiteral Select Many	2
8.	Basic OTHER Offer-Accept Constant	5
9.	Basic OTHER OpenQuestion-Answer One Object mandatory fields	5
Basic Relation Rules		
10.	Basic OTHER Switch	1
11.	Basic Background	1
12.	Basic Sequence	1
13.	Basic OTHER IfUntil Tree	1
14.	Basic OTHER Elaboration	1
15.	Basic OTHER Condition Then	2
16.	Basic Title	3
17.	Basic OTHER IfUntil Tree Then	3
18.	Basic SELF Alternative	3
19.	Basic Ordered Joint	4
Total Matches:		39

Table 6.3: Alternatively Applicable but Not Fired Basic Transformation Rules for Rendering the Vacation Planning Desktop GUI

1.	Basic Background small	1
2.	Basic SELF Alternative small	3
3.	Basic OrderedJoint small	4
Potential Total Matches:		8

Figure 6.15 shows the start screen of the customized final desktop GUI. For customizing this GUI, we spent approximately 5 hours on defining the Vacation Planning transformation rule set and 8,5 hours on creating the desktop CSS.

6.2.3 GUI Customization for Multiple Devices

We applied the approach presented in this doctoral dissertation for customizing GUIs for two devices. In particular, we rendered another GUI for a Samsung Galaxy Nexus smartphone after having created the Vacation Planning Desktop GUI.

For customizing the Desktop GUI, we applied our iterative and incremental interaction design process including the customization through transformation rules (see Figure 5.5), including customizations through an application specific style sheet (i.e., desktop CSS). For customizing the

Table 6.4: Generic-custom Transformation Rules fired for Rendering the Vacation Planning Desktop GUI

Generic-custom Rules		*Matches*
1.	OTHER Informing const confirmation	1
2.	OTHER Informing oneEObject withoutAttributeLabels	1
3.	OrderedJoint notSplit	2
4.	SELF Alternative notSplit	2
5.	SELF Alternative split	3
6.	Background verticalLayout	6
Total Matches:		15

Table 6.5: Application-specific Custom Transformation Rules fired for Rendering the Vacation Planning Desktop GUI

Custom Layout Rules		*Matches*
1.	AlternativeTopLevel	1
2.	AlternativeNavigation	1
3.	Offer zurueck	6
Custom Adjacency Pair Rules		
4.	Informing Verfuegbarkeit	1
5.	Informing UnterkunftDetails	1
6.	Informing ArtikelDetails	1
7.	Informing Zimmerdetails	1
8.	Informing VeranstaltungDetails	1
9.	Informing AnreiseDetails	1
10.	Informing Suchergebnisse	5
11.	OpenQuestion Search	1
12.	ClosedQuestion Raeumlichkeit	1
13.	ClosedQuestion Artikel	2
14.	ClosedQuestion Veranstaltung	3
15.	ClosedQuestion Ankunft	3
16.	ClosedQuestion Unterkunft	3
Total Matches:		32

smartphone GUI, we applied our process for iterative GUI customization (see Figure 5.14 above), with the Application-tailored Device Specification shown in Table 6.6 and using the "Screen-based Device Tailoring with Vertical Scrolling" strategy for tailoring the GUI.

We adapted rules from the generic-custom and the custom rule sets and achieved a tailored GUI after 4 automated tailoring loops. It is noteworthy that we did not have to specify new transformation rules, but rather adapt the matching and firing of existing transformation rules through adaptations in their LHSs. We illustrate these adaptations below through presenting the matched and fired rules for each rule set and comparing them to the rules matched and fired when rendering for the Desktop device.

Table 6.7 shows that 37 Communication Model patterns where matched by 19 different *basic* transformation rules that were fired for rendering the smartphone GUI. This is the same number of transformation rules, but two patterns less in comparison to the Desktop GUI. The *basic*

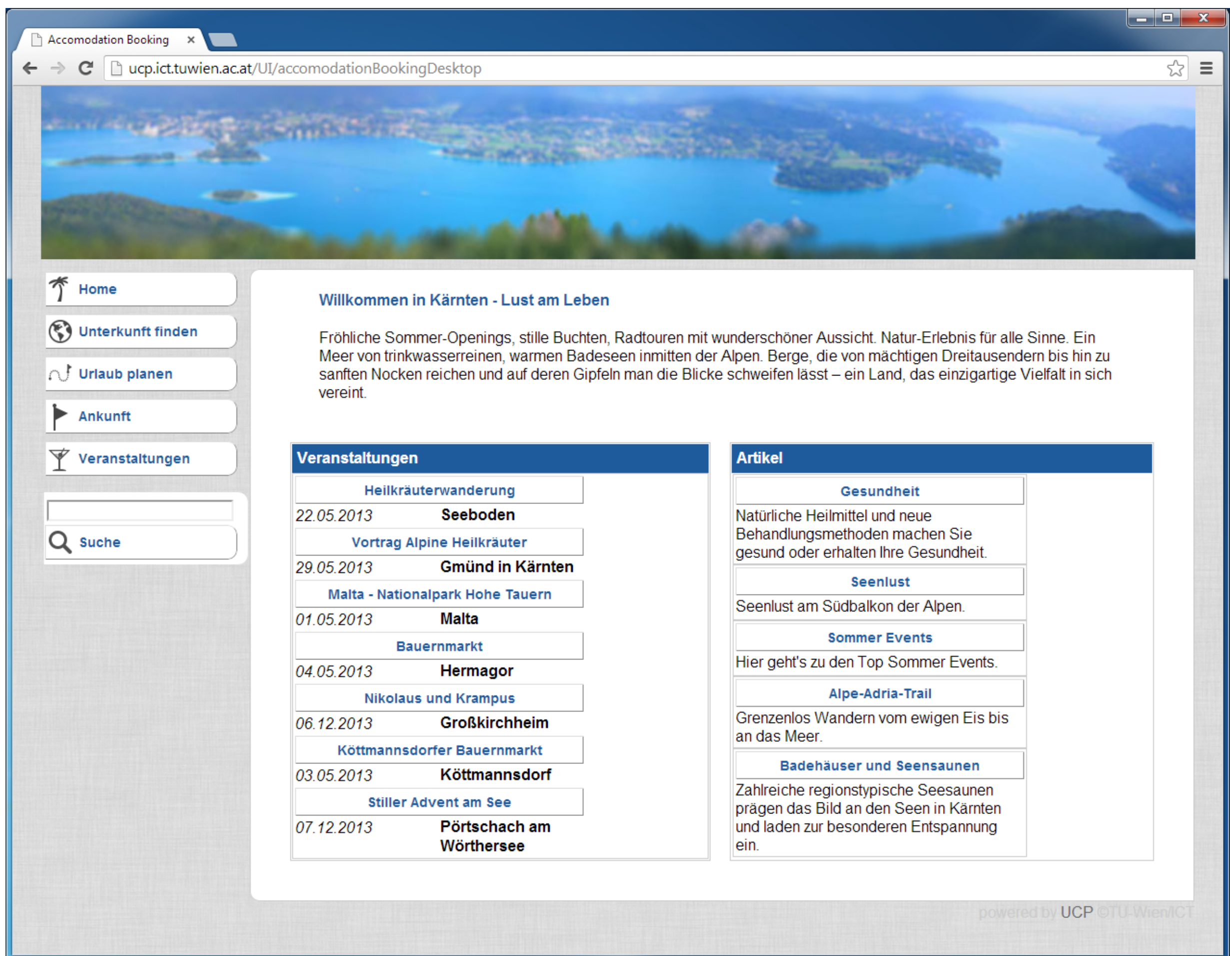


Figure 6.15: Home Screen of Customized Desktop GUI for Vacation Planning

Communicative Act and the *basic* Adjacency Pair rules are matched to the same number of Communication Model patterns. The difference lies in the *basic* Relation Rules. In comparison to the Desktop GUI, the *Basic Background small* rule (shown in row 11 in Table 6.2) fired instead of the *Basic Background* rule (shown in row 11 of Table 6.7). The *Basic SELF Alternative* rule fired only once instead of 3 times and the *Basic SELF Alternative small* rule fired once instead of never (see rows 13 and 14 in Table 6.7). This time also *basic alternative* rules were fired, because the first GUI did not fit and an optimal GUI was found after 4 automated tailoring loops.

Achieving an optimal GUI only after four loops means that rules that alternative transformation rules were used. So, *basic minimal* rules that were not fired when generating the optimal GUI in loop 4 may have been fired in the previous 3 loops. The alternatively matched rules from the *basic* rule set are shown in Table 6.8. This table contains the not fired *Basic Background* rule and one match for the *Basic SELF Alternative* rule. Instead their *basic alternative* rules (see row 11 and 14 in Table 6.7) fired due to our automated tailoring. The *Basic SELF Alternative small* and *Basic OrderedJoint small* rules provide rendering alternatives for the fired rules shown in line 13 and 19 of Table 6.7.

Overall, Table 6.8 lists 4 *basic* rules for 7 Communication Model patterns, which means that the total number of possible GUIs is $2^7 = 128$. This means that one more custom rule matched a Communication Model pattern that was previously matched by two *basic* rules when rendering for

Table 6.6: Smartphone Samsung Galaxy Nexus Device Specification

name	Smartphone Samsung Galaxy Nexus
resolution_x	360 (i.e., 720 with scaling factor 2)
resoltuion_y	640 (i.e., 1280 with scaling factor 2)
dpi	96
defaultCSS	smartphone.css
pointingGranularity	coarse
toolkits	HTML
scrollWidth	1
scrollHeight	5

the desktop. Table 6.9 shows that the 18 Communication Model patterns were transformed by 7 different transformation rules. In comparison to the Desktop GUI, the *SELF Alternative notSplit* and the *SELF Alternative split* rules are each fired once more and the *OTHER Elaboration verticalLayout SatTop* rule was fired instead of the *Basic OTHER Elaboration* rule. These rules compensate for two not fired rules from the *basic* rule set and one not fired rule from the custom rule set, shown in Table 6.10.

The custom rule set for the Smartphone GUI contains the same rules as for the Desktop GUI, but only 15 rules were fired for 31 Communication Model patterns. This is one pattern and one transformation rule less than for the Desktop GUI. The reason is that the *SELF Alternative split* rule of the generic-custom rule set was fired instead of the custom *AlternativeTopLevel* transformation rule (shown in row 1 in Table 6.5) to the TopLevel Alternative A1 shown in Figure 6.14.

Figure 6.16(b) below shows the start screen of the customized final Smartphone GUI. For customizing this GUI we spent approximately 0,5 hours on adapting the custom transformation rules (both generic and application-specific) and 1 hour on adapting the Desktop style file. No further effort was required for creating the Smartphone GUI, as the back-end was already available from the development of the Desktop version.

In comparison to the Desktop GUI (5 hours spent on rule development), the amount of time spent on customization through transformation rules was much smaller, because no new rules had to be created. Also the adaption of the corresponding style sheet (8,5 hours) required less time. Both show that *reuse* of customizations is supported in our development process. Furthermore, the overall development effort decreases if GUIs for multiple devices have to be developed, because the back-end is already available from the GUI developed before.

The complete GUIs of our Vacation Planning application are accessible in the Web, both for Smartphone[1] and Desktop[2].

6.3 Evaluation of (Semi-)automatically Generated GUIs

This section presents the evaluation results of (semi-)automatically generated GUIs for three different applications. We put "semi" in parentheses because we evaluated fully-automatically

[1] `http://ucp.ict.tuwien.ac.at/UI/accomodationBookingSmartphone`
[2] `http://ucp.ict.tuwien.ac.at/UI/accomodationBookingDesktop`

Table 6.7: Basic Transformation Rules fired for Rendering Vacation Planning Smartphone GUI

Basic Communicative Act Rules		*Matches*
1.	Basic SELF Request Constant	1
2.	Basic OTHER Accept Constant	1
Basic Adjacency Pair Rules		
3.	Basic SELF AdjacencyPair	1
4.	Basic OTHER Offer-Accept One Object	1
5.	Basic OTHER Informing One Object	1
6.	Basic OTHER OpenQuestion-Answer One Object	2
7.	Basic OTHER ClosedQuestion-Answer Many EnumLiteral Select Many	2
8.	Basic OTHER Offer-Accept Constant	5
9.	Basic OTHER OpenQuestion-Answer One Object mandatory fields	5
Basic Relation Rules		
10.	Basic OTHER Switch	1
11.	*Basic Background small*	*1*
12.	Basic Sequence	1
13.	*Basic SELF Alternative*	*1*
14.	*Basic SELF Alternative small*	*1*
15.	Basic OTHER IfUntil Tree	1
16.	Basic OTHER Condition Then	2
17.	Basic Title	3
18.	Basic OTHER IfUntil Tree Then	3
19.	Basic Ordered Joint	4
Total Matches:		37

Table 6.8: Alternatively Applicable but Not Fired Basic Transformation Rules for Rendering the Vacation Planning Smartphone GUI

1.	Basic Background	1
2.	Basic SELF Alternative	1
3.	Basic SELF Alternative small	1
4.	Basic OrderedJoint small	4
Potential Total Matches:		7

generated GUIs and GUIs that were generated automatically and customized manually. So, manual customization is optional in our approach.

First, we evaluated a fully-automatically generated Desktop and a Smartphone GUI for our *Bike Rental* trial application. Through evaluating the fully-automatically generated GUIs, we wanted to evaluate their usability. For evaluating the usability of the GUIs, four usability experts performed a Heuristic Evaluation using both of them. The presentation of this Heuristic Evaluation is based on [RWP+13].

Second, we used a simple *Flight Booking* application for evaluating two different device-tailoring strategies for small devices, e.g., smartphones, through user studies. In particular, we evaluated three different GUI versions for this application on an iPodTouch and a Motorola Xoom Tablet PC. To avoid distractions of the participants through basic usability problems, David

Table 6.9: Generic-custom Transformation Rules fired for Rendering the Vacation Planning Smartphone GUI

Generic-custom Rules		*Matches*
1.	OTHER Informing const confirmation	1
2.	OTHER Informing oneEObject withoutAttributeLabels	1
3.	*OTHER Elaboration verticalLayout SatTop*	*1*
4.	OrderedJoint notSplit	2
5.	*SELF Alternative notSplit*	*3*
6.	*SELF Alternative split*	*4*
7.	Background verticalLayout	6
Total Matches:		18

Table 6.10: Application-specific Custom Transformation Rules fired for Rendering the Vacation Planning Smartphone GUI

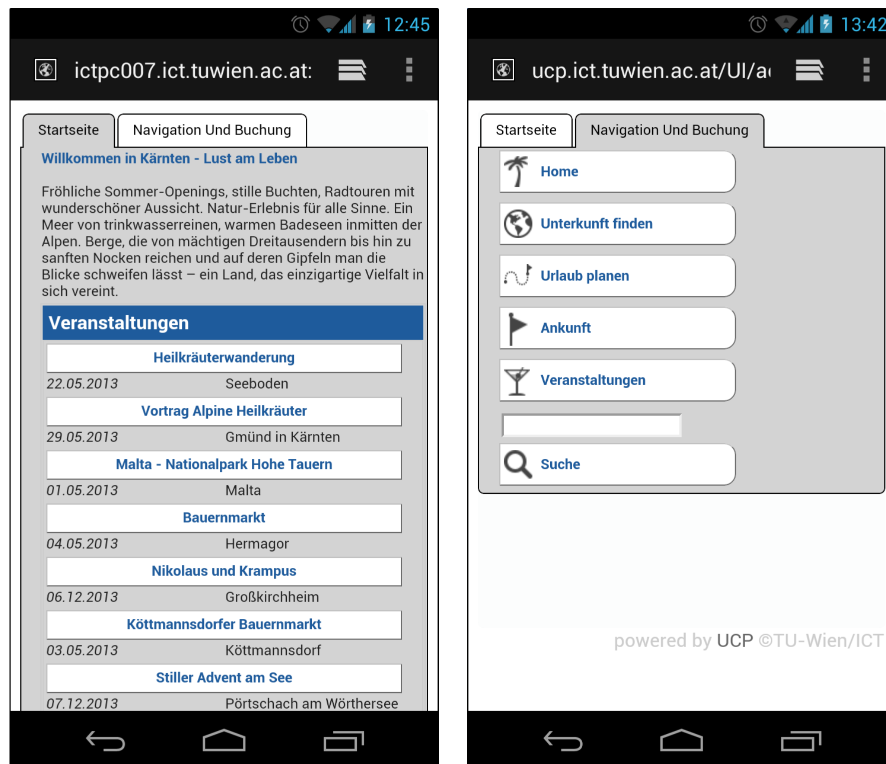
Custom Layout Rules		*Matches*
1.	AlternativeNavigation	1
2.	Offer zurueck	6
Custom Adjacency Pair Rules		
3.	Informing Verfuegbarkeit	1
4.	Informing UnterkunftDetails	1
5.	Informing ArtikelDetails	1
6.	Informing Zimmerdetails	1
7.	Informing VeranstaltungDetails	1
8.	Informing AnreiseDetails	1
9.	Informing Suchergebnisse	5
10.	OpenQuestion Search	1
11.	ClosedQuestion Raeumlichkeit	1
12.	ClosedQuestion Artikel	2
13.	ClosedQuestion Veranstaltung	3
14.	ClosedQuestion Ankunft	3
15.	ClosedQuestion Unterkunft	3
Total Matches:		31

Alonso-Ríos conducted a Heuristic Evaluation and we customized the GUI according to its results before actually starting with the user studies. The presentation of these user studies is based on [RARP+13, RPAR+13, ARRP+14].

Third, we iteratively developed and customized a Desktop and a Smarphone GUI for our *Vacation Planning* application and used them, and the corresponding real-world Web-site as a basis for a user study. Prior to the user study, four usability experts performed a Cognitive Walkthrough and we customized the GUIs based on its results to achieve a high-level of usability. This user study has been conducted as part of the GENUINE project[3] at the University of Salzburg[4] and its summarized presentation here is based on [WMFT13, BFS+13].

[3] `http://genuine.ict.tuwien.ac.at`

[4] more precisely at the Human-Computer Interaction & Usability Unit at the Center for Advanced Studies and Research in Information and Communication Technologies & Society (ICT&S Center) of the University of Salzburg



(a) Vacation Planning Smartphone Home Screen. (b) Vacation Planning Smartphone Menu Screen.

Figure 6.16: Vacation Planning GUI Displayed on a Samsung Galaxy Nexus Device.

6.3.1 Evaluating Fully-Automatically Generated GUIs using Bike Rental

To evaluate the quality of fully-automatically generated GUIs, we performed an informal case study, in which we created a Bike Rental trial application and generated both a smartphone and a desktop GUI fully-automatically. Subsequently, both GUIs were evaluated independently by four usability experts. This Heuristic Evaluation was conducted as part of the GENUINE project[5] at the University of Salzburg[6] and led by Barbara Weixelbaumer. Its results are presented here based on [RWP+13].

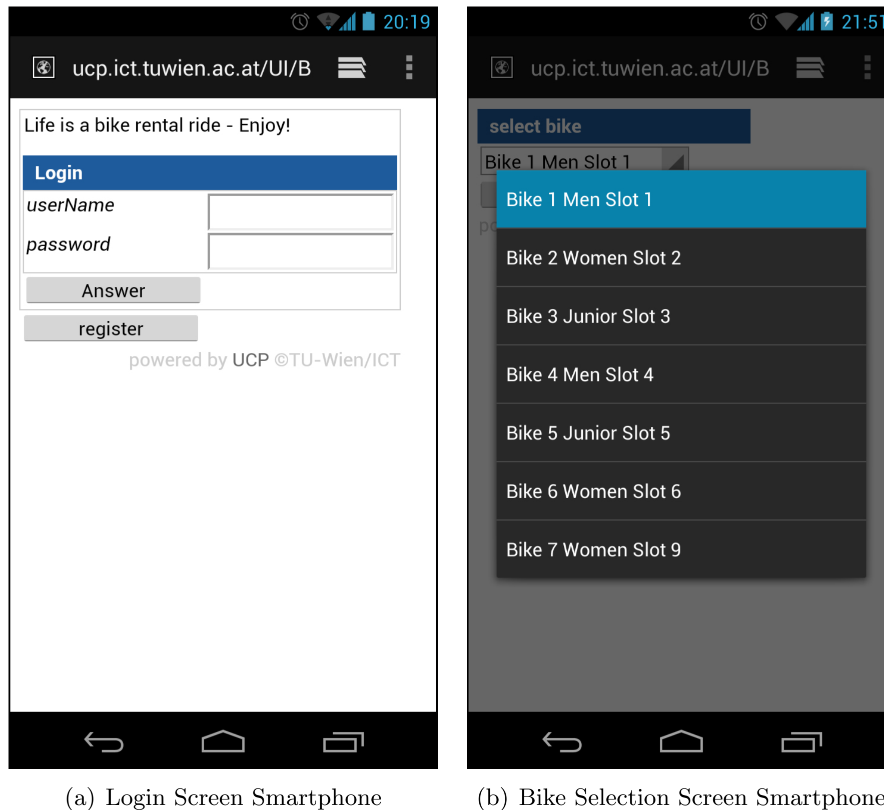
At the beginning of our case study, we selected the usability guidelines to apply. Such guidelines are, for example, available in the literature [Sch00], or provided by big software companies to ensure a good level of usability for their applications (e.g., [App12, Mic12]). Such company guidelines typically contain characteristics that make the applications of a certain company recognizable and are thus only to a limited extent generally applicable. In research there is no need to brand applications, which is why scientific usability principles are of a more generic nature (e.g., Nielsen's heuristics [Nie93]). Today, a huge body of knowledge on usability exists, as well as a plethora of heuristics and guidelines. Though each set of heuristics or guidelines uses specific wording, they cover similar usability features [Nas12].

[5] for more information on the project please refer to `http://genuine.ict.tuwien.ac.at`

[6] more precisely at the Human-Computer Interaction & Usability Unit at the Center for Advanced Studies and Research in Information and Communication Technologies & Society (ICT&S Center) of the University of Salzburg

An interesting concept in the context of usability is gender-inclusive design. "Gender-inclusive" means that the usability of an application should be equally good for all users, regardless of their sex, age, and other characteristics. Guidelines for gender-inclusive GUI design can be used to achieve this goal and to avoid the discrimination of certain users. The set of gender-inclusive design guidelines presented in [WMW+12] for example, provides guidelines for interaction designers, UI design guidelines, programming guidelines, guidelines for policy makers and booking application specific guidelines. A closer look at these guidelines shows that they cover similar features as established usability heuristics (e.g., [Nie93, Nas12]) and, more importantly, that they do not contradict each other.

We based our case study on Nielsen's heuristics [Nie93] and the guidelines for gender-inclusive GUI design [WMW+12]. The difference between heuristics and guidelines is very subtle, because both can be used to specify similar usability principles. For our informal case study, we keep the different terminology, because Nielsen refers to very general principles that can be considered in various ways, whereas the gender-inclusive design guidelines offer more concrete hints.



(a) Login Screen Smartphone (b) Bike Selection Screen Smartphone

Figure 6.17: Automatically Generated Smartphone Screens [RWP+13]

Our case study uses the Bike Rental trial application to investigate the impact of considering the selected usability heuristics and guidelines in the context of automated model-driven GUI generation. The Bike Rental application is a research prototype and supports renting and returning bikes for registered users, as well as creating a user account (i.e., registering a new user). Customization in model-driven GUI generation is possible through the definition of additional application-specific transformation rules, or the modification of models involved in the transformation approach. We decided that customization during the transformation process and in

subsequent iterations was out of scope, because we investigated the impact of considering usability heuristics/guidelines in the first iteration of the model-driven development process. Our rationale was that a higher level of usability of automatically generated GUIs requires fewer manual customizations in subsequent development cycles.

6.3.1.1 Setup and Evaluation Method

We considered the usability heuristics defined by Nielsen [Nie93] and the gender-inclusive design guidelines defined in [WMW+12] while developing the Bike Rental application, and used them as a basis for the Heuristic Evaluation of the generated GUIs. In particular, we aimed to consider them while developing the Discourse-based Communication Model and in the design of the *basic* transformation rule set [RPK13b], which supports automated generation of device-tailored GUIs.

Discourse-based Communication Models specify all possible flows of interaction and thus the navigation in a clear and structured way. Hence, we considered heuristics and guidelines that are related to the flow of interaction during their creation. Examples are Nielsen's heuristic "Flexibility and efficiency of use" and the gender-inclusive design guideline "Define the navigation clearly".

The transformation rules map Communication Model patterns to GUI patterns. Communication Models define the interaction between the system and a user, but they do not specify any style or layout information. This information is added through the transformation rules during the transformation process, in particular, the transformation rules refer to styles defined in a Cascading Style Sheet (CSS). So, the transformation rules directly influence the interaction on the widget level and the visual appearance of the GUI. Examples of heuristics and guidelines that were considered in the transformation rule design are Nielsen's heuristic "Aesthetic and minimalist design" and the gender-inclusive guideline "Place recurring widgets consistently".

Our informal case study investigated the impact of considering usability guidelines in the context of fully-automatic GUI generation. For our evaluation **setup**, we generated two HTML GUIs for the Bike Rental application: one for a desktop PC (with a screen resolution of 1200×1024 pixels, 96 dpi), and one for a smartphone (Samsung Galaxy Nexus with a resolution of 720×1280, 315 dpi). Figure 6.17 shows two screen shots of the smartphone HTML GUI.

Heuristic Evaluation is an efficient **method** of expert evaluation, even for early stage prototypes in interface engineering life-cycles [Nie94]. Heuristic Evaluation allows for identifying usability problems with relatively low financial and temporal efforts, while still guaranteeing a valuable outcome. The evaluation is carried out independently by a small set of user interface design specialists – the *evaluators* – and the whole evaluation process is organized and coordinated by a *supervisor*. Each evaluator is provided with the same usage scenarios and set of heuristics/guidelines. In particular, Nielsen postulated 10 heuristics for such evaluations [Nie94]. By running through these scenarios the evaluators examine the user interface and check its compliance with the provided usability principles. Each violation of the heuristics/guidelines is documented with a detailed description of the deficit and the heuristic/guideline it violates. Subsequently, the supervisor compiles the evaluations results and creates a list of identified usability problems. This list is handed back to the evaluators for a severity rating of each usability problem and analyzed subsequently by the supervisor.

The supervisor of our Heuristic Evaluation instructed four evaluators to apply Nielsen's 10 usability heuristics [Nie94] and a set of 11 gender-inclusive design guidelines [WMW+12] for each

prototype (i.e., smartphone and desktop GUI) on each device (i.e., smartphone and desktop PC). Thus, each evaluator had to conduct four evaluations, with a reversed order of the GUIs under evaluation. Two evaluators were female, two male and all four had expertise in Human-Computer Interaction, interface design and Heuristic Evaluations. The evaluators were instructed to go through the three scenarios (registering a new user, renting a bike, returning a bike) and to fill-in prepared evaluation sheets. The experts were given several days to perform the test, so that they were able to conduct their evaluation concentrated and without hurry.

Next, the supervisor compiled a list with all detected problems based on the evaluation sheets that she received and returned this list to the evaluator for the severity rating. The evaluators used the scheme presented in Table 6.11 for this rating.

Table 6.11: Severity Levels for Usability Problems

0:	I do not agree that this is a usability problem at all.
1:	Cosmetic problem only; does not have to be fixed unless extra time is available.
2:	Minor usability problem; fixing should be given low priority.
3:	Major usability problem; important to fix, should be given high priority.
4:	Usability catastrophe; imperative to fix before product can be released.

Finally, the supervisor analyzed the severity ratings and rated the severity for each violation.

6.3.1.2 Results

This subsection presents the results of the Heuristic Evaluation, starting with the results when using Nielsen's heuristics, continuing with the results of the gender-inclusive design guidelines and finally comparing them with each other.

The left part of Table 6.12 shows the *10 usability heuristics postulated by Nielsen* [Nie93], together with how often these heuristics were violated through the application. The right part shows how severe these heuristic violations were rated by the evaluators, presenting the median for each guideline.

Table 6.12: Usability Heuristics According to Nielsen (based on [RWP+13])

Nielsen Heuristics		*Violations Count*			*Severity Rating (Median)*		
Nr.	Heuristic	Smartphone	Desktop	Total	Smartphone	Desktop	Total
1	Visibility of the system status	5	7	12	3.5	3	3
2	Match between system and the real world	6	7	13	3	3	3
3	User control and freedom	6	7	13	4	3	4
4	Consistency and standards	7	13	20	3	3	3
5	Error prevention	6	5	11	3	3	3
6	Recognition rather than recall	2	1	3	3	3	3
7	Flexibility and efficiency of use	3	3	6	3	3	3
8	Aesthetic and minimalist design	4	1	5	2	1	1.5
9	Help users recognize, diagnose and recover from errors	2	2	4	4	4	4
10	Help and documentation	2	3	5	3	3	3
	Total Violations	43	49	92			

The evaluation according to Nielsen resulted in a total of 49 violations for the desktop and 43 for the smartphone GUI. Heuristic number 4 (i.e., consistency and standards) was violated most often (13 times on desktop, 7 times on smartphone). We conjecture that the flow of interaction and certain interface elements of the prototype are not consistent with common applications of

similar functionality. Examples for such violations are “automated log-out after registration” or “horizontal layout of Web-forms” on the Desktop GUI (see Figure 6.18). These violations are partly caused by the prototypical character of our application (i.e., violations in the interaction) and partly result from the use of our *basic* transformation rules (i.e., style and layout violations). These rules are designed in a consistent, but aesthetically rather minimalistic way, because they have been designed without a certain application in mind. Furthermore, the evaluators remarked problems related to the layout (e.g., horizontal orientation of Web-forms). Automated layout creation, especially of large screens (e.g., desktop) is a difficult problem, which seems only solvable through manual intervention of the designer [KL10]. UCP:UI supports the specification of Layout Hints in transformation rules [RPV12]. In this case study, however, we did not apply Layout Hints when generating the Bike Rental GUIs, because their inclusion requires the creation of application-specific custom rules and thus manual customization.

Figure 6.18: Automatically Generated Desktop Screen with Horizontal Form Layout

The second most frequently violated heuristics were number 2 (i.e., match between system and real world) and 3 (i.e., user control and freedom), with 7 violations for the desktop and 6 for the smartphone version each. Violations assigned to heuristic number 2 indicate that the system does not use concepts or language familiar to the user (e.g., “ID” is not user language). These problems result from missing information in the fully-automatic generation process. The requested information cannot be specified in high-level interaction models or general-purpose transformation rules (as it is application-specific). Hence, such problems (e.g., text customization or the explicit declaration if a field shall conceal input values) can only be solved through the definition of application-specific transformation rules or manual customization of the generated GUI, which were both out of scope of this case study. Examples of violations of heuristics number 3 are: “Back” button is missing, editing user data is not possible or “cancel” leads to logging off the user. The violation of this heuristic implies that the interaction specified in the Discourse-based Communication Model does not match the user’s needs or expectations.

It is noteworthy that heuristic number 6 (i.e., recognition rather than recall) was violated three times in total only (once on the desktop and twice on the smartphone version). We relate this to the clear or even (too) restrictive design of the interaction model and the minimalistic design of the transformation rules, which facilitate user orientation.

Our results show that the violation frequency of a heuristic does not directly correlate with its severity. For example, heuristic number 9 (i.e., help users recognize, diagnose and recover from errors) is violated only twice on each device, but judged with the highest severity level. Severity ratings were determined for all heuristics. Violations have been judged as rather severe, as the

total median for 9 out of 10 heuristics is 3 or higher, and have approximately the same level on both devices (for details see Table 6.12).

The left part of Table 6.13 shows *11 gender-inclusive design guidelines* [WMW+12], together with how often they were violated on different screens of the application for the smartphone and the desktop GUI. The right part shows how severe these violations were rated by the evaluators, presenting the median for each guideline.

Table 6.13: Gender-inclusive Design Guidelines (based on [RWP+13])

Gender-inclusive Design Guidelines		*Violations Count*			*Severity Rating (Median)*		
Nr.	Guideline	Smartphone	Desktop	Total	Smartphone	Desktop	Total
1	Provide a navigation bar (especially on mobile devices)	1	1	2	2	2	2
2	Use intuitive and consistent wording	4	7	11	2.5	3	3
3	Make clear which input is processed with which action	3	4	7	4	3.5	4
4	Define the navigation clearly	3	7	10	3.5	3	3
5	Make sure that all UI widgets work correctly on all supported devices	0	0	0	-	-	-
6	Provide price information	4	4	8	4	4	4
7	Place recurring widgets consistently	0	0	0	-	-	-
8	Place Menus on the left side of the UI	0	0	0	-	-	-
9	Use images	5	6	11	3	3	3
10	Less is more	0	0	0	-	-	-
11	Avoid scrolling on desktop UIs	0	1	0	0	3	3
	Total Violations	20	30	50			

In total, the desktop application violated the gender-inclusive heuristics 30 times, the smartphone application caused 20 violations. The most frequently violated guidelines were number 2 (i.e., use intuitive and consistent wording) and number 9 (i.e., use images), with 11 violations in total each. Such violations are, e.g., that “answer” is not an intuitive label for a log-in function (for guideline 2), “bike pictures are missing” and “boring visual design” (for guideline 9). Guideline number 4 (i.e., define the navigation clearly) was violated 10 times (7 on desktop, 3 on mobile), indicating that the navigational structure is not adequate. Examples for usability problems that the evaluators assigned to this guideline are: missing feedback (e.g., error messages) or log-out is not consistently available. Violations of guideline number 6 (i.e., provide price information) were reported 4 times on each device. These violations relate again to the prototypical character of the application in general (e.g., no price information) and of the interaction model in particular (e.g., missing feedback).

It is noteworthy that the application with fully-automatically generated GUIs did not violate 4 out of 11 guidelines at all (i.e., 5 – make sure all widgets work correctly, 7 – place recurring widgets consistently, 8 – place menus on the left side of the UI, 10 – less is more). With regard to these guidelines, the restrictive interaction and the minimalistic but consistent design of the transformation rules were beneficial.

For the gender-inclusive design guidelines it is the same as for Nielsen’s heuristics, the number of violations does not directly influence the severity. For example, guideline number 6 is ranked on the fourth place according the violations count, but was rated as the most severe violation. Severity ratings were determined for 7 guidelines. Violations have, again, been judged as rather severe in general, as the total median for 6 out of these 7 guidelines is 3 or higher, and have approximately the same level on both devices (for details see Table 6.13).

The *results of the evaluation* show that more violations were identified with respect to Nielsen’s heuristics, though indicating that the gender-inclusive design guidelines constitute a valuable addition. The gender-inclusive guidelines support the identification of usability problems that are not covered by Nielsen’s heuristics (e.g., missing price information), although there is a

certain overlap. For example, the gender-inclusive design guideline 7 ("Place recurring widgets consistently") concretizes Nielsen's guideline 4 ("Consistency and Standards").

In order to investigate how easy these guidelines are to apply in the context of a Heuristic Evaluation, the study supervisor provided a feedback questionnaire to the evaluators. All four evaluators rated the gender guidelines as *easy* to apply on a 4 point Likert-scale (1 – very easy, 2 – easy, 3 – hard, 4 – very hard). Nielsen's heuristics were rated as *very easy* to apply by three of the experts, giving "being already familiar" with these heuristics as a reason for their answer. The fourth expert rated the evaluation with Nielsen's heuristics as hard, arguing that they are more abstract and general than the gender-inclusive guidelines. We (the authors of [RWP+13]) conjecture that performing a step-wise evaluation, based on Nielsen's heuristics and the gender-inclusive guidelines, is an appropriate way to achieve a high level of usability, when evaluating for a wide range of people with a diverse socio-demographic background.

The severity of a certain violation does not seem to depend on a certain device, as it has been rated about the same on the smartphone and on the desktop GUI. The severity of usability problems can be used to prioritize them.

It turned out that the violated heuristics and guidelines did not differ much between smartphone and desktop PC, which is understandable as both GUIs are based on the very same interaction model and have been generated firing rules from the same *basic* set of transformation rules. Nevertheless, there are slightly fewer violations of the smartphone GUI (63) compared to the desktop GUI (79). We conjecture that this is due to the automatically generated layout, which is easier to accomplish on a the smartphone, as it is more restricted (in terms of screen size) and thus results in fewer possibilities to place widgets, if scrolling should be avoided.

Finally, it is noteworthy that the usability violations partly resulted from "confusing" application behavior. These violations result from an inadequate high-level interaction model and not from the model-driven transformation process. The low level of usability can, therefore, not be attributed to model-driven UI generation alone.

Abstracting our informal case study results presented above from our concrete experiences gained with the Bike Rental application, we can formulate the following lessons learned:

- Usability problems regarding the "behavior" of an application are related to inadequate high-level interaction models and cannot be attributed to the transformations.
- Application-specific information is required to achieve a high level of usability.
- Manual customizations is required to achieve a high level of usability.

We fixed the detected usability problems in the customized smartphone version of our Bike Rental GUI.[7] We conjecture that this customized version is better than the automatically generated version in terms of usability, as it does not have the problems revealed by the Heuristic Evaluation of the fully-automatically generated GUIs. Strictly speaking, we would have had to evaluate the customized GUI again, because we could have introduced new usability problems through our customizations, but creating the "perfect" bike rental application is out of scope for this work.

[7] `http://ucp.ict.tuwien.ac.at/UI/BikeRental`

6.3.2 Evaluating Device-tailoring Strategies for Small Devices using Flight Booking

Web-based usability guidelines typically agree that unnecessary scrolling [Mic10], and in particular horizontal scrolling [Bev05], should be avoided on desktop devices. Recently published guidelines for touch-based devices, in contrast, state that scrolling is part of such a device's user experience and, therefore, not bad at all [App12]. The different user experience originates in the more "natural" swiping gesture, in comparison to dragging a scroll-bar with a mouse, or turning its scroll-wheel. These guidelines, however, do not contain hints on whether vertical scrolling is preferable to tabs, horizontal or two-dimensional scrolling, which we investigated in our user study.

Comparable user studies have been performed in the past, but only with desktop devices[8] or cell-phones [BFJ+01] available at the time. Previous user studies on touch-based devices (e.g., PDA or smartphones) assumed that the user does not like to scroll and, therefore, primarily focused on comparing strategies to avoid or minimize scrolling. According to [JJM+05], users prefer vertical over two-dimensional scrolling on small screens.

We used a simple Flight Booking scenario for our user study. In this scenario the user selects a departure, a destination airport and enters a travel date first. Next, the application computes a list of flights from which the user selects one. Finally the user is asked to enter personal details (name and birthday) and payment information (credit card number, expiration date, etc.).

We specified this scenario with a Discourse-based Communication Model and rendered three GUIs with different layouts for it. The first GUI was render for the Motorola Xoom Tablet, resulting in what we refer to here as *horizontal scroll-based layout (H-UI)*. The second GUI was rendered for an iPodTouch Device (equivalent to iPhone 4 or below, which shows the GUI in the same way) using the "Screen-based Device Tailoring without Scrolling" strategy, resulting in what we refer to here as *tab-based layout (T-UI)*. The third one was rendered again for the iPodTouch, using the "Screen-based Device Tailoring with Vertical Scrolling" strategy, resulting in what we refer to here as *vertical scroll-based layout (V-UI)*.

The horizontal scroll-based layout (H-UI) was rendered for the tablet, but it did not have to be tailored because it fit the available display size. So, we compare a non-tailored GUI (H-UI) with two GUIs tailored differently for the iPodTouch in fact. All these GUIs are given in the form of Web pages on the usual browsers of each device. The tablet device used has a resolution of 1280×800px and the iPodTouch 320×480px.

6.3.2.1 Heuristic Evaluation

The Heuristic Evaluation preceding our user study was conducted by David Alonso-Ríos based on the usability taxonomy presented in [ARVGMRMB10]. The detected usability problems that we fixed before starting the user study where primarily style adaptations (e.g., widget background colors or fonts) and adaptations of text labels to avoid misunderstandings (e.g., "origin" and "destination" instead of "from" and "to") and facilitate orientation (e.g., clear screen names including consecutive numbers). These problems affected and were fixed on all three GUI versions under evaluation. For further details on this Heuristic Evaluation please see David Alonso-Ríos' PhD thesis [AR14].

[8] http://www.surl.org/usabilitynews/51/paging_scrolling.asp

6.3.2.2 User Study

The purpose of our study was to evaluate different ways of tailoring a GUI for a touch-based device. In our *setup*, we basically combine quantitative data (i.e., task completion time and error rate) with questionnaires to collect subjective data like in [JMMN+99, BFJ+01]. In particular, we designed our user study similarly to [WdMFP11], who also investigated touchscreen usability (in the context of clicking on hyperlinks). Our user study investigates the correlation between the time a user needs for a usual task on a given screen, depending on how the screen is laid out and we collected subjective opinions on several usability-related questions through questionnaires.

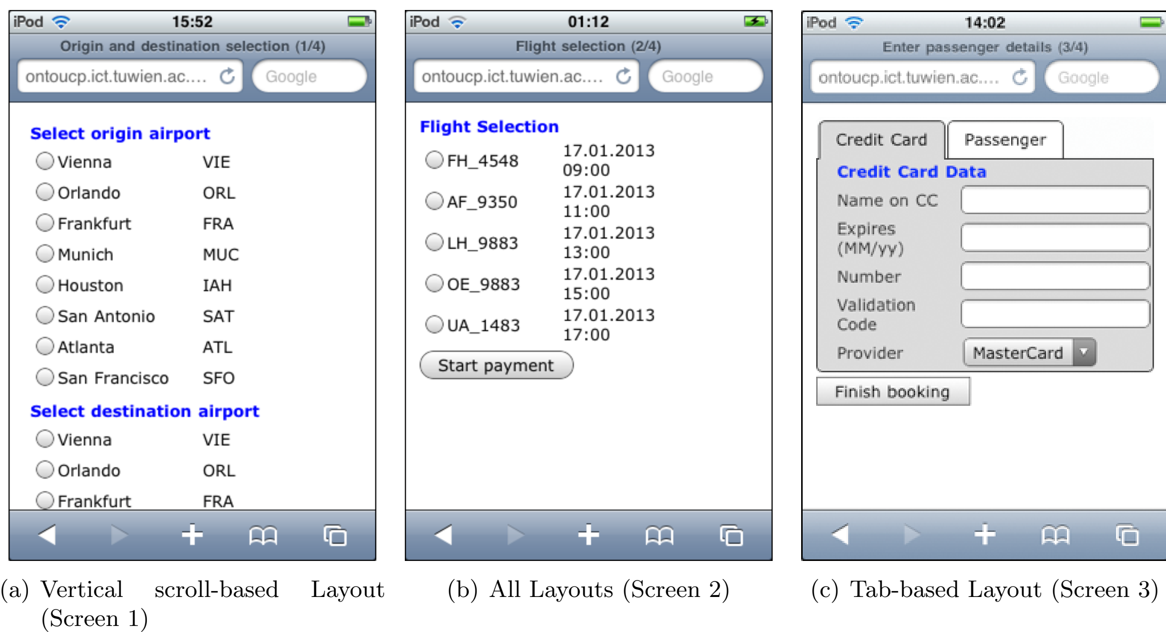
For accomplishing the task of booking a flight, we provided GUIs with 3 screens each:

Screen 1 presents a form for the selection of the departure and destination airports and the date.

Screen 2 presents a list of available flights for the selection of one of them.

Screen 3 requires the user to enter the given billing information.

Figure 6.19 shows the three screens of the Flight Booking application. Screens 1 and 3 can be seen in tailored versions in Figure 6.19(a) and Figure 6.19(c). Screen 2 showed the list of flights and fits the screen on each device. Therefore, the layout of Screen 2 is identical in the two tailored and the non-tailored GUIs.



(a) Vertical scroll-based Layout (Screen 1)

(b) All Layouts (Screen 2)

(c) Tab-based Layout (Screen 3)

Figure 6.19: Screens tailored for iPodTouch / iPhone [RPAR+13]

The non-tailored GUI fits PC and tablet device screens, but not the one of the smartphone. On the latter, there is a kind of tunnel view on **Screen 1** (as shown in Figure 6.20) and on **Screen 3**. Note, that this is incidentally a horizontal scroll-based layout. Since the browser does not indicate that there is more information on the right, however, unless the screen is being touched, this tunnel view can be very confusing, although it is one-dimensional.

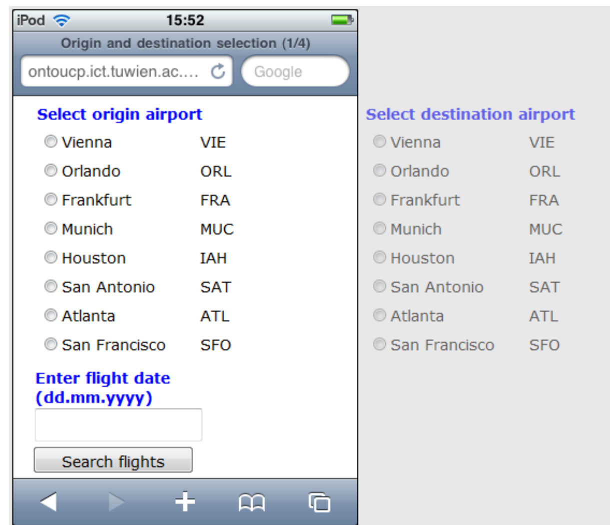


Figure 6.20: Tunnel View of the Whole Page — Horizontal scroll-based Layout (H-UI) [RPAR+13]

We conducted three different test runs in our study to compare these device-tailored GUIs (tab-based and vertical scroll-based layout, respectively) and the non-tailored GUI:

1. Run 1: Tab-based (T-UI) vs. horizontal scroll-based layout (H-UI);
2. Run 2: Vertical (V-UI) vs. horizontal scroll-based layout (H-UI);
3. Run 3: Vertical scroll-based (V-UI) vs. tab-based layout (T-UI).

For the first run we used two different touch-devices (an iPodTouch and a Motorola Xoom). For the other two runs, we only used the iPodTouch, because for the Motorola Xoom in both cases all the information would be displayed without scrolling at all.

We hired 60 **participants** of about the same age (between 18 and 39 years old, with an average of 26 years) and of about the same level of education (students and two assistants with doctoral degrees). 68.3% of the participants were regular users of mobile devices and 28.3% of the participants had used these kinds of devices previously but never owned one. 23.3% of the participants reported to book flights on the Internet regularly, 33.3% booked only a few flights previously, and 43.3% never booked a flight on the Internet before.

20 users participated in run 1 (i.e., 10 users for the iPodTouch and 10 users for the Motorola Xoom), 10 users participated in run 2 and 30 users participated in run 3. For each run, we split the test users into two groups. Both user groups had to perform the task twice with both versions of the GUI under test, on either smartphone or tablet, one group in reversed order to avoid learning effects. Much as the gender distribution in the faculty (for electrical engineering) where the participants have been hired, only few females participated. In the first test run, there was only 1 female involved and in the second one none. In the third test run, 2 females participated in one group and 1 in the other.

At the beginning of each experiment, we informed the users about the content of the study and the **procedure**. We emphasized that we did not want to test them, but the usability of the GUIs that they were going to use, so that they would act confidently and relate problems rather to the application and not to themselves. We also asked them to fill in some demographic data and to

specify their English level and familiarity with smartphones. We did this to avoid bias through basic understanding or device-handling problems. Finally, we asked them for their consent to film the screen of the iPodTouch for subsequent video annotation and to record the final interview, both without filming their faces. Only the hands were visible while they operated the iPod and we only needed the audio track to extract data from the final interviews. We furthermore fixed the iPod on the table, so that they had to stay in portrait mode and was not able to switch to landscape mode. We did so, because the tailored GUIs have been developed for portrait mode and we could not have filmed the screen of the iPod clearly visible if the users would have held the iPod in their hands, which would have made the video annotation a difficult task.

Before starting the video camera, we explained them the following scenario, leaving a print-out version in front of them, so that they did not have to remember the data:

> "Imagine it is Tuesday 14.02.2012, 11:55am and your boss Mr. Huber tells you to book a flight ticket for his wife as quickly as possible. Mrs. Huber is already waiting at the airport!
>
> Book a flight from *Munich* to *Atlanta* on *02/14/2012* at *1pm* for Mrs. *Anna Huber* (*age 47*). Pay for it using her husband's (Max Huber's) *VISA Credit Card* with the number: *1258 8569 7532 1569* (validation code: *354*) and the expiration date *12/14*."

We especially emphasized the urgency of the task, so that the user would concentrate on the task and finish it as quickly as possible. The task specification contains all the information that needs to be entered during the task to avoid bias through different data.

Each participant then completed the given task twice, according to the defined runs. After the participants finished these tasks, we collected their subjective opinions on several usability-related issues through a questionnaire and finally asked them if they had further comments, which we taped on video for later evaluation.

Analysis. In our study we evaluated quantitative (tasks completion time and error rate) and qualitative data (subjective questionnaires). An independent variable for our statistical analysis was the GUI with the three different values H-UI, V-UI and T-UI. This variable is dichotomous as we only compared two GUIs per run. A second independent variable was the order of presenting the GUIs to the participants, e.g., T-UI first and then H-UI, or vice versa. The dependent variables were (adjusted) task completion time or error rate, respectively. For the *(adjusted) task completion* analysis, we measured the total time a participant spent on a screen for solving the given task. In addition, we measured the time needed for text input (on the keyboard), the time for loading the screen and submitting information, and the time for validation and error messages, since these are independent of the specific GUI used and may bias the results due to irrelevant random effects. Therefore, we subtracted all these times from the total time, resulting in *adjusted task time*, which includes orientation on the screen, widget selection with fingers, and value selection from widgets.

For the *error rate* analysis, we measured the number of errors each participant made on a certain screen. Errors in our study occurred in cases where the user hit the submit button before entering the required information on a certain page.

In all test runs, we were primarily interested in the correlations between the task time / error rate and the type of user interface used. Due to the three runs in our study design, the type of the GUI is dichotomous and the calculated time is on an interval scale. Therefore, we calculated the

point-biserial Pearson correlation coefficients. Our tests also showed that the variables are nearly normally distributed and thus satisfy the prerequisites of the point-biserial Pearson correlation. Note, the difference to the more common t-test is that the correlation covers *coherence*, while the t-test deals with *differences of means*.

All tests and interviews were taped on video to facilitate the data extraction. We used the video-annotation tool Anvil[9] to precisely extract task completion time and error rate, which we subsequently converted to Comma Separated Value (CSV) lists and put into SPSS to perform the statistical analysis.

After using two different GUIs, the participants were asked to compare the relative usability of both GUIs rather than assess each one independently, through a subjective usability questionnaire. We consulted the USE Questionnaire[10], the W3C's WAI[11], the Software Usability Measurement Inventory[12] (SUMI), and the Cognitive Dimensions framework [BBC+01]. Our questions as given in Figure 6.21 were based on these questionnaires and primarily on the usability taxonomy by Alonso-Ríos et al. [ARVGMRMB10], with some adaptations. In particular, we phrased the attributes as specific questions, which we tried to make short and self-explanatory. The last question was a general one about the user's overall assessment. For each question, the users had to state which GUI they preferred (on a Likert scale with preference being "extreme", "strong", "moderate", and "equal").

- Which interface makes information more visible?
- Which interface makes interaction more intuitive?
- Which interface makes it easier to figure out what to do next?
- Which interface makes it clearer how to use it?
- Which interface lends itself more to how you like to work?
- Which interface requires less manual interaction?
- Which interface demands less precision on your part?
- Which interface demands less time from you?
- Which interface makes interaction more efficient?
- Which interface is more visually attractive?
- Overall, which interface would you use to book a flight?

Figure 6.21: Questions from our Questionnaire [RPAR+13]

The subsequent paragraphs present the results of our study for each run. In particular, we present the analysis of *adjusted task time* and *error rate* together with an evaluation of the subjective questionnaires. Figures 6.22(a), 6.22(b), 6.23 and 6.24 summarize the questionnaire results graphically. The x-axis shows the values of the Likert scale and the z-axis the percentage of how many participants selected a certain value. The y-axis shows the 11 different questions that we asked. We extracted these questions and showed them in Figure 6.21, because they were unreadable in the diagram. The questions are related to the lines through their order (first question, to line one, second question to line 2, etc.) and through color coding.

Run 1: Tab-based vs. Horizontal Layout

The results of our first run have been published in [RPAR+13] and are summarized here based on this publication.

[9] http://www.anvil-software.org/

[10] http://www.stcsig.org/usability/newsletter/0110_measuring_with_use.html

[11] http://www.w3.org/WAI/EO/Drafts/UCD/questions.html

[12] http://sumi.ucc.ie/en/

20 users participated in the first test run. 10 users performed the task on a Motorola Xoom tablet device and 10 users on an iPodTouch.

Adjusted Task Time Analysis. As presented in our study setup, we calculated the Pearson correlation between adjusted task time for each screen at its layout. Tables 6.14 and 6.15 show the resulting correlations for the iPodTouch and the Xoom tablet, respectively. Positive correlations mean that the GUI with the tab-based layout performed better, whereas negative correlations indicate a better task performance with the horizontal scroll-based layout.

Table 6.14: Run 1: Correlation between Adjusted Task Time on a Screen and its Layout on an iPod-Touch [RPAR+13]

	Pearson Corr.	Sig. (1-tailed)	*T-UI* av. time	*H-UI* av. time
Time Screen1 × UI	**0.52**	0.01	14.23s	29.57s
Time Screen2 × UI	0.12	0.31	8.96s	9.98s
Time Screen3 × UI	**0.37**	0.05	8.09s	13.34s

Table 6.15: Run 1: Correlation between Adjusted Task Time on a Screen and its Layout on a Motorola Xoom [RPAR+13]

	Pearson Corr.	Sig. (1-tailed)	*T-UI* av. time	*H-UI* av. time
Time Screen1 × UI	**-0.56**	0.01	13.37s	7.43s
Time Screen2 × UI	-0.17	0.23	7.45s	6.38s
Time Screen3 × UI	**-0.66**	0.01	8.00s	4.57s

On the iPodTouch, one can see that users performed better with the tab-based layout tailored for this device, since there are significant correlations (p-value = 0.05) on Screen 1 and Screen 3 (see Table 6.14). Screen 3 shows a lower correlation probably because the user saw on this screen that additional information is available on the right in the non-tailored horizontal scroll-based version. Screen 2 almost shows no correlation since all information fitted on the screen and was immediately available in both versions. In fact, Screen 2 was identical for both tested GUIs, and therefore, we ignore the resulting data for Screen 2, here and in all other experiments.

The tables also show the data of completion times minus time for text input, etc., see above (in seconds). Table 6.14 shows that the sum for all screens is on average for the tab-based UI (T-UI) 31.28s and for the horizontal scroll-based UI (H-UI) 52.88s, on the iPodTouch.

In Table 6.15 shows that we got negative correlations on the Motorola Xoom tablet, implying that the horizontal scroll-based layout (H-UI) performed better than the tab-based layout (T-UI). The adjusted task time using the tab-based layout (T-UI) is significantly (p-value = 0.05) higher than using the horizontal scroll-based layout (H-UI) for Screens 1 and 3. This means that the user performed better on the H-UI on the tablet. The reason is that all information fitted onto the screens, so that no scrolling was needed on this device. However, in the tab-based layout additional clicks were needed. The sum of times for all screens is on average for the tab-based UI (T-UI) 28.82s and for the horizontal scroll-based UI (H-UI) 18.37s.

Error Rate Analysis. Table 6.16 shows the correlation between the number of errors and the used User Interface on the iPodTouch, together with average number of errors for each screen. The tab-based UI leads to significantly (p-value = 0.05) fewer errors on the first screen. The positive prefix of the Pearson correlation means that the users performed better with the tab-based GUI. On the second screen there was no error at all, so no correlation could be calculated. On the third screen the correlation is not significant.

Table 6.16: Run 1: Correlation between the Number of Errors on a Screen and its Layout on an iPod-Touch [RPAR+13]

	Pearson Corr.	Sig. (1-tailed)	*T-UI* av. errors	*H-UI* av. errors
Errors Screen1 × UI	**0.65**	0.01	0.2	1.7
Errors Screen2 × UI	-	-	0	0
Errors Screen3 × UI	-0.33	0.08	0.2	0

Table 6.17 shows the correlation between the number of errors and the GUI used on the Motorola Xoom. For this device and on all screens, the correlation is not significant.

Table 6.17: Run 1: Correlation between the Number of errors on a Screen and its Layout on a Motorola Xoom [RPAR+13]

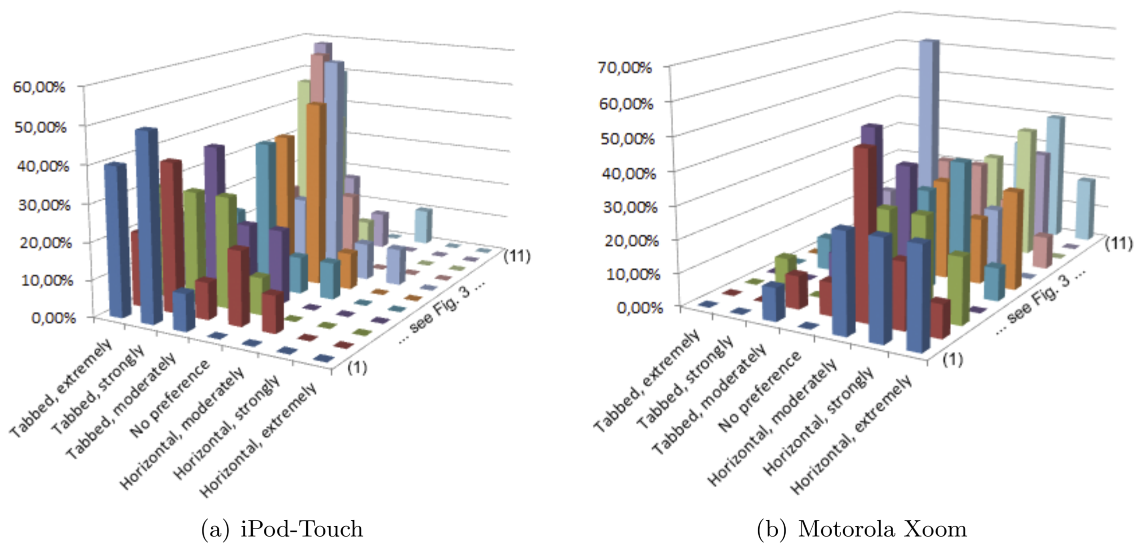
	Pearson Corr.	Sig. (1-tailed)	*T-UI* av. errors	*H-UI* av. errors
Errors Screen1 × UI	-0.229	0.17	0.1	0
Errors Screen2 × UI	-	-	0	0
Errors Screen3 × UI	-0.229	0.17	0.1	0

Subjective Results. Figure 6.22 summarizes the subjective results for Run 1. The subjective results were diametrically opposed depending on the device. For the smartphone, the participants clearly preferred the tab-based UI according to the answers to most questions. In their overall assessment, a total of 90% of the participants preferred ("strongly" or "extremely") the tab-based UI, whereas only 10% preferred ("moderately") the horizontal scroll-based UI. Between 80% and 90% of the participants thought that the tab-based UI lent itself more to how they like to work, and that it was clearer to use, more visually attractive, and more efficient to use. The users did not perceive nearly as much difference regarding manual effort, but they were clearly aware that operating the horizontally-scrolling UI was more time-consuming.

For the tablet PC, in sharp contrast to the opinions for the smartphone, 90% of the participants preferred the horizontal UI in their overall assessment. In fact, participants preferred (almost always overwhelmingly) the horizontal UI over the tabbed UI in all categories except visual attractiveness (for which 50% of them preferred the tabbed UI and 40% the horizontal UI). Interestingly, even though the tab panels leave a considerable amount of white space on the tablet screen, some users remarked in interviews that they liked the "compact" look of the tabbed UI. Still, the tab-based GUI tailored for the smartphone was no real choice for the tablet PC also in terms of the subjective results.

Result Discussion. In general, the tab-based layout performs better on the iPodTouch. The results of the adjusted task time analysis and the error rate analysis are consistent with the result from the subjective questionnaire.

For the Motorola Xoom, the error rate analysis does not suggest any preference for one of the two GUIs. The adjusted task time analysis, however, shows a better performance of the non-tailored GUI. Also the analysis of the subjective questionnaire is consistent with this result. So, the non-tailored GUI performs better on the Motorola Xoom. The fact that all the information is shown on the non-tailored GUI without scrolling is a possible explanation for this preference.



(a) iPod-Touch

(b) Motorola Xoom

Figure 6.22: Run1: Summary of Subjective Questionnaire Results [RPAR+13]

Run 2: Vertical Layout vs. Horizontal Layout

The results of our second run have been published in [ARRP+14] and are summarized here based on this publication.

We conducted the remaining two test runs only on the iPodTouch, since all information fitted on the screen of the Motorola Xoom tablet, which would lead to the same results as given above. The second test run has been conducted with 10 participants, testing vertical scrolling versus horizontal scrolling.

Adjusted Task Time Analysis. Table 6.18 summarizes the statistical results of the adjusted task time analysis in Run 2. It shows that the participants performed significantly (p-value = 0.05) better on Screens 1 and 3 of V-UI, since there are positive correlations in Table 6.18 for these screens. Since the correlations are higher than in the first test, this already indicates that vertical scrolling also performs better than a tab-based layout. The sum of times for all screens is on average for the vertical scroll-based UI (V-UI) 23.39s and for the horizontal scroll-based UI (H-UI) 54.24s.

Table 6.18: Run 2: Correlation between Adjusted Task Time on a Screen and its Layout on an iPod-Touch [ARRP+14]

	Pearson Corr.	Sig. (1-tailed)	*V-UI* av. time	*H-UI* av. time
Time Screen1 × UI	**0.74**	0.01	10.20s	36.78s
Time Screen2 × UI	-0.05	0.42	8.03s	7.48s
Time Screen3 × UI	**0.59**	0.01	5.15s	9.97s

Error Rate Analysis. Table 6.19 shows the correlation between the error rate and the two used User Interfaces (V-UI and H-UI), together with the average number of errors for each screen. On the first screen the error rate is significantly (p-value = 0.05) smaller. On the other two screens there was no error at all, so no correlation could be calculated. So, the error rate for the vertical

scroll-based layout (V-UI) is significantly smaller than for the horizontal scroll-based layout (H-UI) for Screen 1.

Table 6.19: Run 2: Correlation between the Number of Errors on a Screen and its Layout on an iPod-Touch [ARRP+14]

	Pearson Corr.	Sig. (1-tailed)	*V-UI* av. errors	*H-UI* av. errors
Errors Screen1 × UI	**0.83**	0.01	0	1.8
Errors Screen2 × UI	-	-	0	0
Errors Screen3 × UI	-	-	0	0

Subjective Results. Figure 6.23 summarizes the subjective results of Run 2. The participants clearly preferred the vertical over the horizontal scroll-based UI according to the answers to most questions. In their overall assessment, a total of 90% of the users preferred the vertical UI. As for the individual usability criteria, users preferred the vertical UI in all categories except for demands on precision.

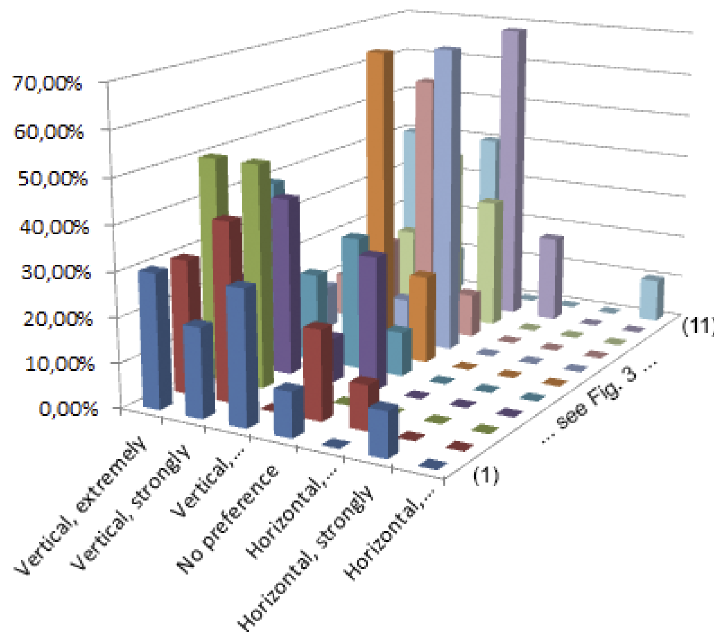


Figure 6.23: Run 2: Summary of Subjective Questionnaire Results - iPod-Touch

Result Discussion. The result of the error analysis is consistent with the result of task time analysis as well as the subjective results, with a clear preference for the vertical scroll-based layout. We conjecture as the major reason for these results, that the horizontal scroll-based layout was *not* fully visible on the smartphone (see, e.g., the “tunnel view” as shown in Figure 6.20).

Run 3: Vertical vs. Tab-based Layout

The results of our third run have been published in [RARP+13] and are summarized here based on this publication.

In the third test run, we compared both device-tailored GUIs with each other to see if the higher correlations of the vertical scroll-based layout in Run 2 are significant. We had 30 participants, again on an iPodTouch only.

Adjusted Task Time Analysis. Table 6.20 summarizes the statistical results for the adjusted task time analysis in Run 3. It shows that the participants performed significantly (p-value = 0.05) better on Screens 1 and 3 of V-UI, since there are positive correlations in Table 6.20 for these screens. The sum of times for all screens is on average for the vertical scroll-based UI (V-UI) 25.81s and for the tab-based UI (T-UI) 36.79s.

Table 6.20: Run 3: Correlation between Adjusted Task Time on a Screen and its Layout on an iPod-Touch [RARP+13]

	Pearson Corr.	Sig. (1-tailed)	*V-UI* av. time	*T-UI* av. time
Time Screen1 × UI	**0.35**	0.01	12.79s	17.98s
Time Screen2 × UI	0.12	0.19	8.17s	9.50s
Time Screen3 × UI	**0.43**	0.01	4.85s	9.31s

Error Rate Analysis. Table 6.21 shows the correlation between the error rate and the two used User Interfaces (V-UI and the T-UI), together with the average number of errors for each screen. For the first and the third screen, the error rate is significantly smaller. On the second screen there was no error at all, so no correlation could be calculated. So, the error rate for the vertical scroll-based layout (V-UI) is significantly (p-value = 0.05) smaller than for the tab-based layout (T-UI) in Screens 1 and 3.

Table 6.21: Run 3: Correlation between the number of errors on a screen and its layout on an iPodTouch.

	Pearson Corr.	Sig. (1-tailed)	*V-UI* av. errors	*T-UI* av. errors
Errors Screen1 × UI	**0.363**	0.01	0	0.23
Errors Screen2 × UI	-	-	0	0
Errors Screen3 × UI	**0.285**	0.01	0	0.2

Subjective Results. Figure 6.24 summarizes the subjective results of Run 3. Overall, the majority (60%) of the participants preferred the vertical scroll-based UI, but 30% preferred the tab-based UI. The answers to the other questions varied much, see Figure 6.24. Therefore, Table 6.22 presents all data illustrated in Figure 6.24 in detail. The vertical scroll-based UI was considered much more intuitive to navigate and to interact with, and slightly less demanding and more efficient to use in general. However, opinions were sharply divided on which GUI made information more visible and which GUI lent itself more to how the users like to work. Interestingly, we were not able to find clear relationships between these diverse answers and the background of participants, e.g., their familiarity with smartphones.

Result Discussion. The result of the error analysis is consistent with the result of the task time analysis as well as the subjective results, with a preference for the vertical scroll-based layout. Only the answers to the question regarding the visual attractiveness are in favor of the use of the tab-based UI.

Discussion of Generality

Let us briefly discuss a few aspects of generality of our study and its results. In particular, the results from a study restricted to only one prototypical and simplified Flight Booking application may be restricted to this domain of application. We claim that our study setup at least mitigates this potential issue for two reasons. First, we defined a scenario that reduced the user task to

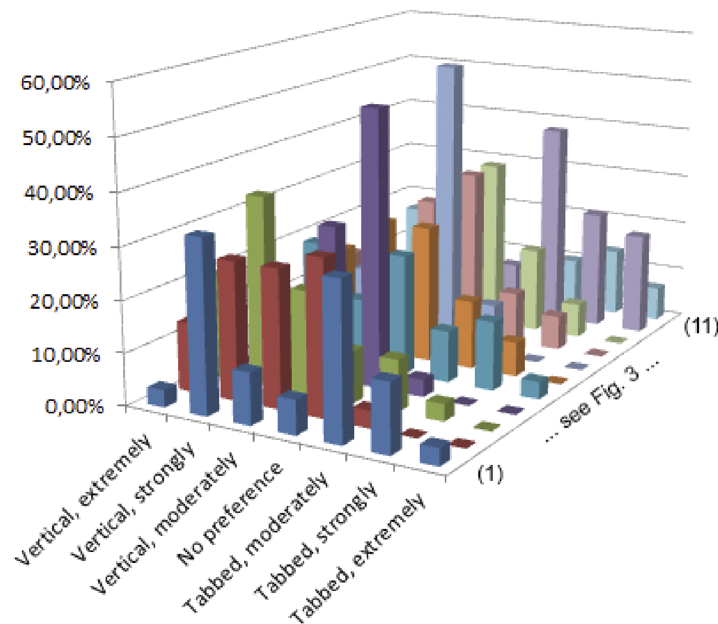


Figure 6.24: Run 3: Summary of Subjective Questionnaire Results - iPod-Touch [RARP+13]

entering pre-given data, without having to decide about actually booking a particular flight or not as usual. Second, we used a very simple prototypical application that allowed only for exactly the interaction that was required to complete the given task, without alternatives. This also helped to keep the user focus on the interaction and allowed all users to complete the task successfully.

The GUIs used in the course of the study were simply HTML-based. They might as well be programmed using any available GUI library, or they may be app GUIs. We conjecture that the essential findings related to size and interaction style would not be much different than the ones we gained with the HTML-based GUIs.

We only used two distinct screen sizes, for the given smartphone and tablet PC. However, current market trends appear to provide a whole spectrum of touchscreen sizes (e.g., notes smartphones, mini pads, etc.). We conjecture that our results hold in general for a certain difference in sizes, also related to the fit of concrete GUIs given. An open question is how good the fit has to be.

Overall, the results of our user study show that users performed better on device-tailored GUIs. In Run 3 we additionally showed that users prefer vertical scrolling over tabs. These findings are reflected in our optimization objectives and the tailoring strategies. In particular, objective 1 (e.g., maximum use of the available space) is consistent with our findings in Run 1 and we provide a tailoring strategy that allows for vertical scrolling up to a given length, which reflects our findings in Run 3.

6.3.3 Evaluating Iteratively Developed and Customized GUIs using Vacation Planning

We used our Vacation Planning application for testing the usability of iteratively developed and customized GUIs. As this application is based on a real-world Web-site, we were able to conduct a comparative user study and evaluate the usability of our semi-automatically generated GUIs (one for a smartphone and one for a desktop device) in comparison to the Web-site. This user study

Table 6.22: Subjective Questionnaire Results: Run 3

Question	*V-UI* preferred				*T-UI* preferred		
	extreme	strong	moderate	equal	moderate	strong	extreme
Which interface makes information more visible?	3.33%	33.33%	10%	6.67%	30%	13.33%	3.33%
Which interface makes interaction more intuitive?	13.33%	26.67%	26.67%	30%	3.33%	0%	0%
Which interface makes it easier to figure out what to do next?	20%	36.67%	20%	10%	10%	3.33%	0%
Which interface makes it clearer how to use it?	0%	13.33%	30%	53.33%	3.3%	0%	0%
Which interface lends itself more to how you like to work?	13.33%	23.33%	13.33%	23.33%	10%	13.33%	3.33%
Which interface requires less manual interaction?	6.67%	20%	26.67%	26.67%	13.33%	6.67%	0%
Which interface demands less precision on your part?	3.33%	13.33%	16.67%	56.67%	10%	0%	0%
Which interface demands less time from you?	13.33%	10%	26.67%	33.33%	10%	6.67%	0%
Which interface makes interaction more efficient?	10%	6.67%	26.67%	33.33%	16.67%	6.67%	0%
Which interface is more visually attractive?	0%	0%	6.67%	10%	40%	23.33%	20%
Overall, which interface would you use to book a flight?	16.67%	20%	23.33%	10%	10%	13.33%	6.67%

was conducted as part of the GENUINE project[13] at the University of Salzburg, led by Barbara Weixelbaumer and Verena Fuchsberger.[14] To evaluate and potentially improve the usability of our initially generated and customized GUIs, usability experts of the University of Salzburg performed a Cognitive Walkthrough before the user study was conducted. The Cognitive Walkthrough was led by Nicole Mirnig at the University of Salzburg and was also performed in the context of the GENUINE project. The author of this doctoral dissertation was only involved to a limited extent in the Cognitive Walkthrough and in the design phase of the user study. The results of both are summarized here briefly based on the corresponding deliverables of the GENUINE project (D5.2 for the Cognitive Walkthrough [WMFT13] and D5.3 for the user study [BFS+13]), because they contribute to the evaluation of this dissertation.

6.3.3.1 Cognitive Walkthrough

A Cognitive Walkthrough is an expert-based evaluation technique where usability experts put themselves in the position of a specific user (e.g., given through a so-called "Persona" that characterizes a specific type of user) and test the UI from this user's point of view [LW97]. The Cognitive Walkthrough was conducted independently by four usability experts at the University of Salzburg. In particular, each expert conducted four tasks (two on the Desktop and two on

[13] `http://genuine.ict.tuwien.ac.at/`

[14] more precisely at the Human-Computer Interaction & Usability Unit at the Center for Advanced Studies and Research in Information and Communication Technologies & Society (ICT&S Center) of the University of Salzburg

the Smartphone GUI) and ranked the detected usability problems according to the severity levels shown in Table 6.11.

Table 6.23 shows the number usability problems that were detected in the initial version of each GUI (column 2 for the desktop and column 3 for the smartphone) through the Cognitive Walkthrough, sorted by their severity. These problems partly related to the GUIs and partly to the application back-end. Columns 4 and 5 show how many of these problems were fixed by us through customizations of the GUI and adaptations of the back-end before the user study was performed.

Table 6.23: Usability Problems in initially customized Vacation Planning GUIs (based on [WMFT13])

Severity Level	*Detected* Problems		*Fixed* Problems	
	Desktop	Smartphone	Desktop	Smartphone
4 – catastrophe	10	10	10	10
3 – major	16	10	13	7
2 – minor	9	14	8	11
1 – cosmetic	3	4	3	4
Sum:	38	38	34	32

Overall, 76 problems were detected that were distributed equally between the two GUIs (i.e., 38 problems each). 10 of these problems were rated as usability catastrophes (i.e., severity level 4) on each GUI. For example, it was not possible to refine the search data of a previous search after the search results had been display. We fixed this problem through customizing the Communication Model. In particular, we used the `update` instead of the `get` Action defined in the `basic` ANM that supports this functionality. Another problem was that some text labels and links were not clearly distinguishable. We this problem through adapting the visual appearance of links and buttons in the corresponding CSS. Two problems rated as catastrophe concerned the back-end. For example, the search functionality did not work correctly. Each problem fixed in the back-end was fixed on all devices. Overall, all catastrophic problems on both GUIs were fixed by us before the user study was performed.

The Desktop GUI contained 16 major problems from which 13 were fixed and the Smartphone GUI contained 10 major problems from which 7 were fixed. For example, we customized the visual design of buttons in the corresponding CSS to make them better recognizable as buttons and we extended the search functionality provided by the back-end. We were not able to fix 6 problems on both devices (e.g., sometimes slow performance or the inclusion of a Map widget) in the given time-frame due to technical limitations with regard to our UCP framework.

Furthermore, the Cognitive Walkthrough reported on 9 minor usability problems for the Desktop and 14 for the Smartphone version, from which we fixed 8 for the Desktop and 11 for the Smartphone GUIs. For example, the usability experts suggested to display the number of search results at the top of the subsequent screen instead of at the bottom. We fixed this problem through a new custom transformation rule, which we used when rendering the GUIs for both devices. Another minor problem, was inconsistent data, which we fixed through correcting the data used by the back-end. Again, the one remaining Desktop and the 3 remaining Smartphone problems (e.g., provide an interactive map for the selection of a region instead of a drop-down box) were not fixable in the given time-frame due to technical limitations with regard to our UCP framework.

Finally, we fixed all cosmetic usability problems detected through the Cognitive Walkthrough before the user study started. For example, we customized colors in the corresponding CSS. The

customized GUIs used for the user study where better than the initially customized GUIs. In summary, we fixed all catastrophic and cosmetic problems detected, and reduced the number of major and minor problems from 26 to 6 (major) and 23 to 4 (minor) problems.

6.3.3.2 User Study

This user study was performed in the context of the GENUINE project at the University of Salzburg, with the primary aim to explore gender differences in the use of GUIs. Its presentation here is based on [BFS+13] and summarizes only the results that fit the scope of this doctoral dissertation.

The **setup** of this user study was based on a revised version of the commercial Web-site[15] (only available with a desktop GUI) and the semi-automatically generated (i.e., automatically generated and customized) desktop[16] and smartphone[17] GUIs. The start screen of the commercial Web-site is shown in Figure 6.25. The tasks that the users were going to perform during the study were supported by all GUIs. We primarily focused on comparing the desktop versions, because comparing the generated smartphone GUI to the commercial desktop GUI accessed via a smartphone would have been biased through device-specific problems on the commercial Web-site.

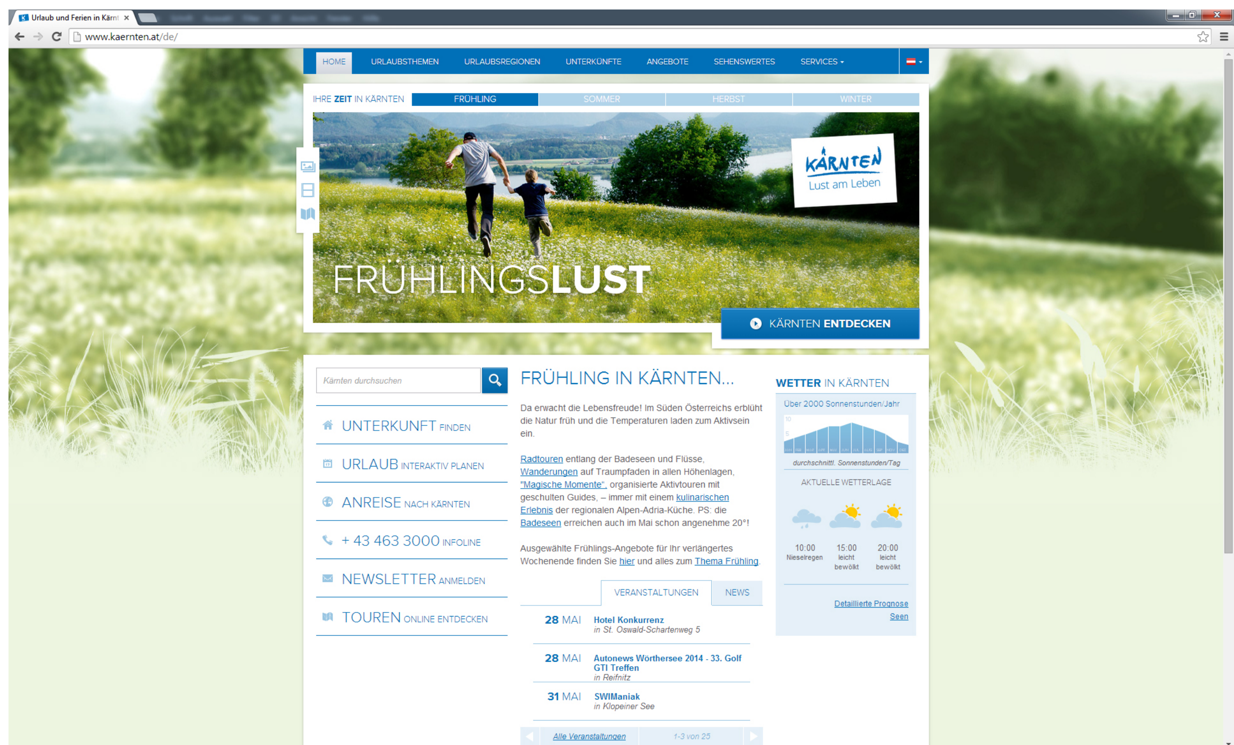


Figure 6.25: Commercial Web-site Start Screen

The user study was conducted with 24 **participants**, which were distinguished through "sex" (i.e., female / male) and "age" (i.e., under 25 / over 40). Based on these two characteristics, four user groups with six participants each were distinguished. Female participants under 25 were

[15]http://www.kaernten.at/
[16]http://ucp.ict.tuwien.ac.at/UI/accomodationBookingDesktop
[17]http://ucp.ict.tuwien.ac.at/UI/accomodationBookingSmartphone

aged between 21 and 25 years (mean = 23) and over 40 between 41 and 54 (mean = 47). Male participants under 25 were aged between 19 and 25 years (mean = 22.7) and over 40 between 46 and 61 (mean = 50.3).

As for the **procedure** of the study (as relevant in the context of this dissertation), all participants first signed an agreement that we were allowed to use the data for scientific purposes and provided socio-demographic data and information on their previous experience with modern technologies and their frequency of use through filling in questionnaires. Subsequently, each participant performed a given task using the commercial Web-site and a similar one using the semi-automatically generated Desktop GUI. We split the participants into two groups, reversing the order in which the two GUIs were tested to avoid learning effects. Both GUIs were rated next through a System Usability Scale (SUS) [Bro96] questionnaire. Finally, each participant performed a third task using the smartphone GUI and again filled in a SUS questionnaire afterwards.

The **analysis** of the results was based on questionnaires, interviews and eye-tracking in addition to the SUS. The SUS is based on a questionnaire with ten questions, where each questions is rated using a five-point Likert scale. Each questionnaire can be mapped to a value between 0 (worst) and 100 (best).[18] In the context of this dissertation we concentrate on the analysis of the SUS, as the other data is out of scope.

Results. Figure 6.26 presents the SUS scores for the commercial Web-site and the semi-automatically generated Desktop GUI. It shows that the generated Desktop GUI was rated clearly better by the 24 users than the commercial Web-site.

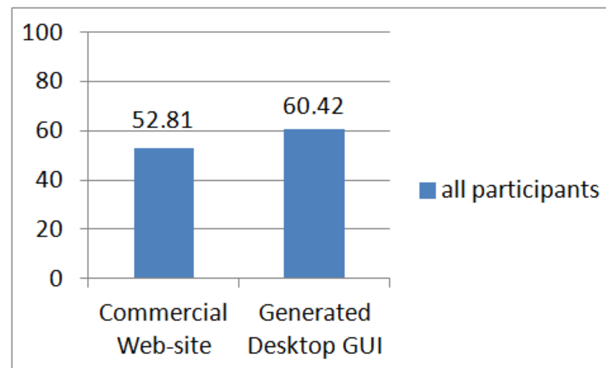


Figure 6.26: SUS Score Commercial and Generated Desktop GUI (based on [BFS+13])

This result is also reflected in the users' answers when asked whether they thought that they had completed a task successfully or not or whether they were not sure. These answers were recorded respectively as "successful", "not successful", or "undecided" and used to measure the so-called *perceived task efficiency*. Figure 6.27 shows the results for both tasks on the commercial Web-site. This figure additionally shows the *real task efficiency* (i.e., did the user really finish the task successfully or not). For the first task, 45.8% of the users (i.e., 11 users) claimed that they had completed the task successfully, whereas only 29.2% (i.e., 7 users) really had. For the second task, 37.5% (i.e., 9 users) claimed that they had completed the task successfully, whereas only 16.7% (i.e., 4 users) really had.

Figure 6.28 shows both the perceived and the real task efficiency for both tasks on the generated Desktop GUI. The perceived task efficiency for both tasks on this GUI is higher and matches

[18] For details on the SUS score calculation please see [Bro96]

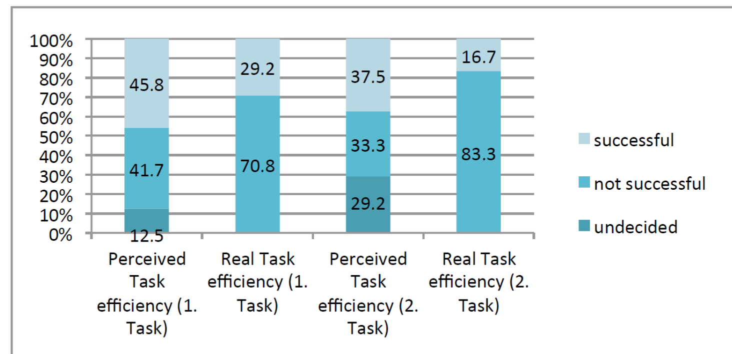


Figure 6.27: Perceived and Real Task Efficiency for Commercial Web-site [BFS+13]

the real task efficiency more closely than on the commercial Web-site. In particular, 62.5% (i.e., 15 users) of the users claimed that they had completed task one successfully, and 58.3% (i.e., 14 users) really had. For task two, 70.8% (i.e., 17 users) claimed that they had completed the task successfully, and 58.3 (i.e., 14) really had.

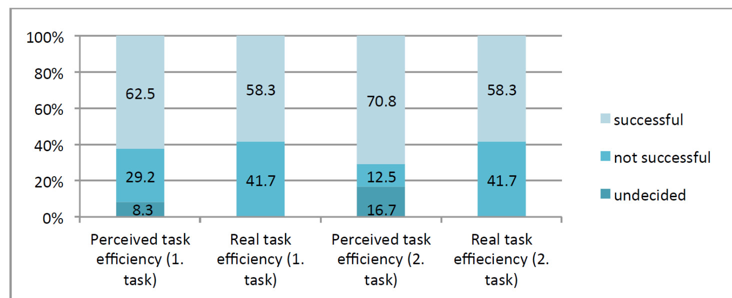


Figure 6.28: Perceived and Real Task Efficiency for Semi-automatically Generated Desktop GUI [BFS+13]

Figure 6.29 shows that the generated Smartphone GUI performed equally well as the generated Desktop GUI in terms of task efficiency. The users performed only one task on this GUI, for which 62.5% (i.e., 15 users) claimed a successful completion and 54.2% (i.e., 13 users) of them really did so. We conjecture that the good performance of this GUI is due to the fact that it has been tailored to the device.

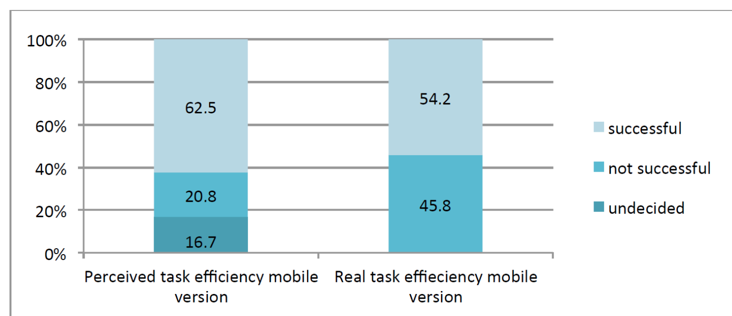


Figure 6.29: Perceived and Real Task Efficiency for Semi-automatically Generated Smartphone GUI [BFS+13]

Comparable data for the commercial Web-site is not available, as this Web-site was not available

as a smartphone version. In particular, we refrained from comparing our mobile version to the commercial Web-site accessed via a smartphone for three reasons. First, this Web-site has not been designed for such a device, which would have led to bias in our data through the device (e.g., through two-dimensional scrolling and widgets that were not designed for touch-based use). Second, not all widgets on this Web-site worked correctly on the smartphone used for our user study (i.e., an iPhone 5). Third, the difference in accessing a previous version of the Web-site through a desktop and a smartphone device was subject to a previous user study in the GENUINE project, which clearly revealed device-related usability problems [WMWT12]. These findings underline the importance of tailoring a GUI for a specific device.

Interestingly, the results of our study were not consistent with the results from the interviews, which were conducted after the users' had completed all tasks. Table 6.24 summarizes the answers to three questions that we selected here to illustrate this inconsistency.

Table 6.24: Interview Results in % and (number of users) based on [BFS+13]

Question	Commercial Web-site	Desktop GUI	Smartphone GUI	Undecided
1. Which GUI has the best navigation?	16.7% (4)	33.3% (8)	20.8% (5)	29.2% (7)
2. Which GUI has the best visual composition?	83.3% (20)	4.2% (1)	4.2% (1)	8.3% (2)
3. Which GUI do you like most?	50% (12)	20.8% (5)	12.5% (3)	16.7% (4)

The answers to the first question underline the SUS scores presented above. In particular, they show that the users preferred the Desktop GUI and even the Smartphone GUI in terms of navigation over the commercial Web-site. The answers to the second question show, however, that the users overwhelmingly preferred the visual composition of the commercial Web-site with 83.3% in contrast to the semi-automatically generated Desktop (4.2%) and Smartphone GUIs (4.2%). These findings are inconsistent with the SUS, but are underlined by the answers to the third question that shows that the commercial Web-site was clearly preferred by the users when asked directly.

The interviews reflect the opinion of the users directly, whereas the SUS scores are calculated based on the answers of ten different questions. The inconsistency between the answers to question three and the SUS score suggests that the visual composition of a GUI is more important for user satisfaction than the ease of navigation. To resolve this inconsistency, either the navigational structure of the commercial Web-site would have to be improved or the visual composition of the generated GUIs.

6.3.4 Summary of Evaluation Results

The Heuristic Evaluation results of the fully-automatically generated Bike Rental GUIs revealed that the consideration of design guidelines allows avoiding basic usability problems, but that iterative development and manual customization are inevitable for achieving a high level of usability. Iterative development facilitates achieving a high-quality interaction model and manual customization is required to achieve the desired "look & feel", because this information cannot be specified in a high-level interaction model per definition.

The results of the user studies using the Flight Booking GUIs showed that users on a small device (i.e., the iPodTouch) performed better on the GUIs tailored this device than on the non-tailored GUI. Moreover, we showed that tailoring a GUI for a small device using vertical scrolling is preferable to fitting the screen through splitting it on touch-based small devices.

The results of the user study using the Vacation Planning GUIs showed that the semi-automatically generated (i.e., customized) GUIs performed better in terms of real and perceived task efficiency than the commercial Web-site. Nevertheless, the users clearly preferred the commercial Web-site when directly asked which GUI they liked better. These results suggest that a good performance in terms of real and perceived task efficiency is not enough to achieve a good level of user satisfaction.

6.4 Key Contributions and Results of this Doctoral Dissertation

This doctoral dissertation accomplished the following four *key contributions* to the field of model-driven GUI generation:

- automated generation of a screen-based GUI model – Screen Model
- (semi-)automatic GUI tailoring
- manual customization
- a tool-supported process for iterative and incremental development

The first key contribution is the automated generation of a screen-based GUI model – the Screen Model. It is a device-dependent but still toolkit-independent GUI model, comparable to a detailed GUI mock-up. It provides a basis for our (semi-)automatic GUI tailoring approach and our support for iterative manual customization.

The second key contribution of this dissertation is our approach for (semi-)automatic GUI tailoring. This approach automatically optimizes the GUI according to our optimization objectives and allows for tailoring a GUI (semi-)automatically (even through choosing between different tailoring strategies) for a given device. Our new approach decouples the high-level interaction model, the transformation rules and the device specification. This facilitates multi-device GUI generation, because a new device specification is necessary to automatically generate a tailored GUI for a corresponding new device. However, further manual customization will still be required to achieve the desired look & feel.

The third key contribution is our support for iterative manual customization of all GUI aspects defined in [Sze96, HME11, PHKB12]. In particular, we support customization through custom transformation rules, Layout Hints, style sheets and different tailoring strategies (see Table 5.1 above). These customizations are persisted, so that they are re-applied in case of re-generation, either through custom transformation rules or through storing them in a separate model. Strictly speaking, we do not support the customization of GUI behavior and adding/removing widgets on Screen Model level. However, we support such customizations through other means (e.g., custom transformation rules) as listed in Table 5.1. This suffices to support customization of all GUI aspects according to [Sze96, HME11, PHKB12], because this work focused on the identification of GUI aspects and not on how their customizations were to be supported or persisted. Supporting such customizations on Screen Model level would entail support for round-trip engineering, which was out of scope for this doctoral dissertation because it is a research topic on its own and subject to future work (see Chapter 8).

The fourth key contribution is an iterative and incremental development process that supports high-quality interaction design and GUI customization for multiple devices. In fact, this process

combines our (semi-)automated tailoring approach and our support for iterative manual GUI customization for multiple devices, based on the Screen Model. The corresponding tool-support is provided through the current version of the Unified Communication Platform (UCP).

Based on these key contributions, we identify three major *results* of this doctoral dissertation:

1. We developed new concepts for interactive GUI development for multiple devices, keeping the designer in the loop and supporting iterative and incremental model-driven GUI development.

2. We implemented our conceptual approach based on a previous version of the Unified Communication Platform (UCP).

3. We evaluated our new approach through applying it for the development of applications with different degrees of complexity (e.g., our Bike Rental and Vacation Planning applications), and we presented the results of usability evaluations of the generated GUIs.

Our evaluation results regarding the application of our approach indicate that model-driven GUI generation may not be cheaper for one device, but that a benefit may be gained if GUIs are required for different devices. For example, in case of our Vacation Planning application we spent considerably less time on developing the smartphone GUI than on developing the desktop GUI. This data is only available for a single application and we have no data regarding the development of the corresponding commercial Web-site, but we conjecture that there will always be a cross-over between manual and model-driven development regarding the development effort, if GUIs for multiple devices are to be provided. At least for our Vacation Planning application, we showed that reuse of specific transformation rules and styles that were created during the customization of a previous GUI is potentially possible to a large extent.

Our evaluation results regarding the usability of the generated GUIs indicate that fully-automatic GUI generation is not sufficient to achieve a good level of usability. So, we conjecture that customization is inevitable to achieve a high level of usability (e.g., comparable to a commercial Web-site). The evaluation results for our Vacation Planning applications also show that providing a good navigation structure is not sufficient to achieve a high level of user satisfaction, but that the visual appearance of the GUI has a major impact.

6.5 How the Results Support the Hypotheses

ad *Hypothesis 1:* “Device-tailored GUIs provide better usability than non-tailored GUIs.”

The answer seems to be clearly yes and there is evidence in literature, for example, for desktop vs. mobile phones [BFJ+01, KR03] and evidence regarding Web-browsing [SST09]. However, scrolling is considered as part of the user experience on touch devices [App12, Mic12], which opens different ways to tailor a GUI to a touch-device. Our interactive GUI generation approach for multiple devices supports different ways of automated tailoring for given device (i.e., vertical- or horizontal scrolling, or no scrolling through page splitting). In a related user study, we used a small Flight Booking application to evaluate these strategies. Our results show that the users performed best on GUIs with vertical scrolling, followed by GUIs that avoided scrolling (i.e., split pages through tabs).

ad *Hypothesis 2:* “Manually customized GUIs provide better usability than fully-automatically generated GUIs.”

Our usability evaluations revealed that all fully-automatically generated GUIs contained usability problems. In particular, we showed that the automatically generated GUIs for our Bike Rental trial application revealed several usability problems. We customized the Bike Rental smartphone GUI based on the Heuristic Evaluation performed at the University of Salzburg.[19] The customized version is better in terms of usability than the automatically generated version, because it has fewer usability problems. The Flight Booking GUIs used for our user study were customized on the basis of David Alonso’s Heuristic Evaluation results and were thus also better in terms of usability than the automatically generated ones. Finally, we already customized the Vacation Planning GUIs during the development of the corresponding interaction design and improved these already customized GUIs based on the results of the Cognitive Walkthrough, again performed at the University of Salzburg. The customized GUIs of all three applications were better than the automatically generated ones, because we fixed the usability problems detected in the Heuristic Evaluations through customizations.

ad *Hypothesis 3:* “Interactive Multi-Device GUI Generation (UCP:UI) supports iterative GUI development.”

Iterative GUI development requires the persistence and reapplication of customizations performed in a previous development cycle. The approach presented in this doctoral dissertation supports the customization of all relevant GUI aspects through a tool-supported process for iterative and incremental interaction model development using automated GUI generation. The automated transformation can be configured through the selection of the most appropriate tailoring strategy for a specific device (i.e., the one that requires the smallest number of customizations) and customizations can be performed and made re-applicable through specific transformation rules (e.g., creating specific widgets or a specific layout) and the specification of a specific styles in a style sheet. Changes that are performed on the Screen Model are stored in a separate model and can potentially be re-applied after the Screen Model has been regenerated.

[19]http://ucp.ict.tuwien.ac.at/UI/BikeRental

7 Discussion

Our experience gained in developing interactive systems and the evaluation results of the corresponding GUIs showed that iterative development and device-dependent GUI tailoring and customization are inevitable to achieve a high level of usability. To better support and to persist such customizations, we provide the Graphical Screen Model Editor (GSME) developed by Alexander Armbruster and the Changes Model. When developing the Bike Renal and the Vacation Planning applications, we used the Screen Model editor to visualize the Screen Model, but we rather chose to perform the customizations through specific transformation rules and style sheet adaptations. So, let us briefly reflect about the reasons for this.

One major reason is that all but style customizations (e.g., which attributes are rendered, layout adaptations) potentially impact the generation of our Screen Model (i.e., our automated device-tailoring). Customization stored in a Changes Model, however, are only re-applied after the Screen Model has been generated, which results in additional effort in comparison to generating the "right" Screen Model right away. Another major reason was that transformation rules can be easily reused (i.e., matched several times when transforming a specific Communication Model, and re-applied/adapted when rendering for another device). The reuse of customizations stored in the Changes Model when rendering a GUI for a different device is not supported by our current implementation of UCP. One minor reason was the maturity of the GSME in terms of usability. The currently available version of the GSME is perfectly suitable for visualizing screens, but it still suffers from usability problems and missing features that make customizations difficult. For example, the direct selection of a container widget to resize is difficult. For the creation of transformation rules, on the other hand, UCP provides an Eclipse-based tree editor, which also allows for checking syntactic correctness of rules [RSKF11]. The RHS of the transformation rule is on CUI level (just like the Screen Model), which we consider the most appropriate one for GUI customization. So, our reasons for performing our customizations through custom transformation rules rather than with the GSME were less effort through direct consideration of customizations during the Screen Model generation, reusability of customizations and better tool-support.

When developing our Vacation Planning application, we found that creating a complex application, even when using model-driven development, is still tedious and error-prone work. We even conjecture that the overall effort is comparable to creating the application manually. However, our experience and data gained through developing GUIs for multiple devices with UCP indicate that the benefit of model-driven development starts with the GUI for the "second" device. The reason is that the models and artifacts (e.g., transformation rules) created in the course of the model-driven development of the first GUI can be reused. The development of our Vacation Planning application showed that also the customization effort for the first device (i.e., Desktop;

5 hours for transformation rules and 8,5 hours for style sheet creation) was higher than for the second device (i.e., Smartphone; 0,5 hours for transformation rules and 1 hour for style sheet creation). Additionally, the development of the final back-end was a lot of effort, which is also only required once for all devices.

Regarding the usability of our generated GUIs, it is interesting that the generated Desktop GUI performed better, but was rated worse than the commercial Web-site. The results of the user study indicate that the visual appearance has a major impact on the user satisfaction of a GUI. These findings were underlined by the ratings of the semi-automatically generated Smartphone GUI, which also performed well, but was received only as well as the semi-automatically generated Desktop GUI. We conjecture that fully-automated generation of a GUI with a high level of usability, using a generic set of transformation rules (e.g., our *basic* rule set) is not feasible at all. One can automatically generate the functionality, but one also has to consider human factors. Therefore, we conjecture that it will be crucial to provide strong tool support for GUI customization, so that even non-engineers can perform them.

Finally, let us briefly reflect about the use of Discourse-based Communication Models in other contexts than GUI generation, which were in principle out of scope for this dissertation. These models are based on Searle's speech act theory [Sea69] and were already used for semi-automatic generation of multi-modal UIs by Dominik Ertl [Ert11]. He showed that such models can be used to semi-automatically generate, e.g., vocal GUIs. Semi-automatically in the context of multi-modal UI generation, however, means that human intervention is required (e.g., for configuring the target speech toolkit), whereas semi-automatically in this doctoral dissertation means that human intervention is optional to improve the usability of the resulting GUIs. Moreover, Discourse-based Communication Models were used as a unified specification for Human-Machine and Machine-Machine Communication by Roman Popp [Pop12]. These dissertations showed that Discourse-based Communication Models could be used in the context of multi-modal UI generation and as unified communication specification, but investigating whether they were more convenient to use than other interaction models was out of scope.

8 Future Work

We identified different topics for future work. An ambitious one is support for GUI behavior customizations and structural GUI modifications that result in Discourse-based Communication Model adaptations/extensions (e.g., adding/removing widgets or creating entirely new screens) through Screen Model modifications. This entails modifications of the Discourse-based Communication Model based on the corresponding customizations performed on the Structural Screen Model and requires the development of new concepts to support round-trip engineering. However, not all customizations can be stored in the Communication Model. Layout and style adaptations, for example, need to be stored elsewhere. In the context of multi-device GUI generation, and in particular their customization, this will require the development of new concepts for propagating such changes directly between the corresponding Screen Models.

Another topic is to provide instances of Communication Model *patterns* (e.g., based on interaction design alternatives [RPK13a]) to better support reuse and to further facilitate the exploration of design alternatives. Such patterns, or more precisely their instances, can be used by the designer as basic building blocks for the creation of new applications as suggested in [SGR+05].

A future topic related to facilitating the exploration of design alternatives is to provide more tailoring strategies. This will involve the development of new configurations for our cost function and their evaluation in the context of different applications, devices and users.

Future support for the generation of native mobile applications (e.g., Android) will facilitate the inclusion of touch-specific widgets in our generation approach and it will facilitate the application of model-driven GUI generation for mobile applications. An increased number of supported GUI toolkits will make our approach more versatile and thus more attractive for multi-device GUI generation.

Finally, we aim to facilitate the collaboration between (research) institutions through facilitating a systematic comparison of state-of-the-art UI generation approaches. To achieve this, we defined a set of 29 criteria, which we called “key-features” (see Appendix A). Facilitating a systematic comparison and connecting researchers to exchange their implementations apart from their concepts should also foster advanced research, because not each institution has to collect the state-of-the-art literature and even more importantly not to implement everything from scratch (partly re-implementing previously published concepts). So far we collected 17 key-feature-based UI generation approach classifications, which were provided by the corresponding researchers who are currently developing a specific approach. Our key feature definition and all currently collected descriptions will be available at a dedicated Web-site[1] (currently under construction) soon.

[1] http://ui-gen.ict.tuwien.ac.at

9 Conclusion

The approach for interactive model-driven generation of graphical user interfaces for multiple devices presented in this doctoral dissertation supports the fully-automated generation of different device-specific GUIs (identified as research problem one in the introduction), through providing four strategies for automated device-tailoring. This facilitates the exploration of design alternatives as different device-tailored GUIs can be generated automatically.

The Heuristic Evaluations of our fully-automatically generated Bike Rental GUIs revealed that manual customization is inevitable to achieve a high level of usability even though usability heuristics/guidelines have been considered explicitly during the interaction model and transformation rule development. So, we defined a tool-supported development process that allows for iterative and incremental interaction design and GUI customization. We evaluated this process through applying it during the development of our Vacation Planning application, which was based on a commercial Web-site and for which we automatically generated and customized a desktop and a smartphone GUI. A subsequent user study that evaluated the generated GUIs in comparison to the commercial Web-site showed that the users' real and perceived task efficiency was better on the generated desktop GUI, which also received higher scores on the System Usability Scale than the commercial Web-site, but was nevertheless rated worse when the users were asked directly which GUI they liked better. The users performed equally well on the smartphone GUI as on the desktop GUI, but rated it again worse than the commercial Web-site when directly asked which GUI they liked better.

We conjecture that achieving a high level of usability is infeasible without customization through a human designer for which this doctoral dissertation provides a sound conceptual basis, with the corresponding implementation based on the Unified Communication Platform. Both have been evaluated based on empirical data, gained through the development of different applications using our new approach, and the subsequent evaluation of the generated GUIs, even in comparison to a commercial Web-site. In terms of multi-device GUI generation, the collected data indicates that the benefit of less development effort through model-driven development can potentially be gained starting from the second device, because most models and artifacts can be reused.

A Key Features of Model-based User Interface Generation Approaches

More than two decades of research on (semi-)automated UI generation have lead to a large body of knowledge and to many (prototypical) implementations [MPV11]. The scope of such UI generation approaches, in terms of which UIs and applications they support to develop, is typically limited and depends on certain properties like run-time or design-time generation, supported modalities or platforms, etc. We modeled a selected set of such properties that allow for the classification of different types of UI generation approaches and called them *key features*. These features provide a common basis for the classification of UI generation approaches, with the aim to facilitate their systematic comparison [Kit04].

Figure A.1 shows a more formal definition of our key features in a UML class diagram. It shows that a `UIGenerationApproach` is characterized through `Properties`. One or more of such Properties are specified through a certain `KeyFeature`.

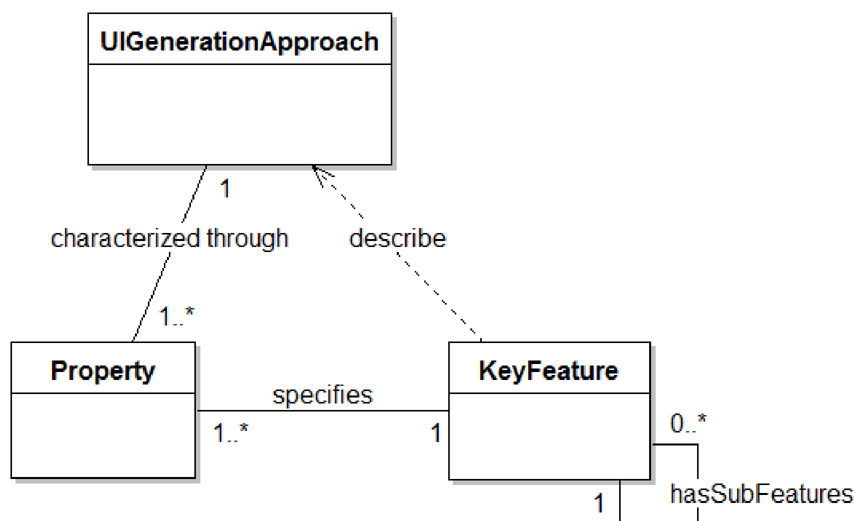


Figure A.1: Key Features

We defined our key features based on literature research, our experience during the development of our own model-driven UI generation approach and its application in various projects and, most importantly, on feedback that we collected from fellow researchers via key-feature based

descriptions of their generation approaches that will be available on our Web-site[1] to facilitate a systematic state-of-the-art research in the future.

State-of-the-art approaches that support the comparison of UI generation frameworks typically concentrate on either the *Models* [LV03, MSO12] involved, the *Method* [SMV07] used to generate the final UI code, the *Tools* that support the method [MHP00] or the final UI [PS12]. These four categories, also referred to as dimensions [Van08], are closely related and the combination of supported features that correspond to these categories is typically unique for a certain approach. Therefore, it is difficult to compare approaches while only considering one of them [HPR12]. The CRF levels and the MDA models can be used to classify the models involved, but we conjecture that from a practical point of view it is also important to know which additional artifacts are required by the generation approach or how the UIs are connected to the application back-end (i.e., application logic and persistence layer). The specification of such properties is supported through our key features, which rely on the CRF levels of abstraction to classify UI models. We additionally introduce features to specify aspects that are not covered by the CRF.

We grouped our key features in five categories: *General* features, features concerning the *Input Models*, the transformation *Method*, the *Tools* and the resulting *Applications* (see Figure A.2).

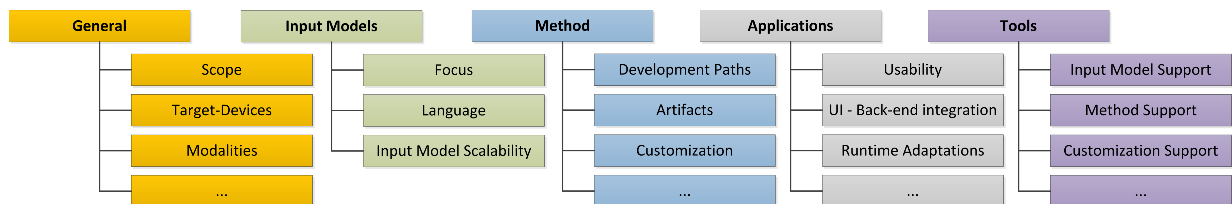


Figure A.2: Key Feature Overview

General features cover basic aspects of UI generation approaches and can be used to roughly classify them. *Input Model* features focus on the models that are required as a starting point for the generation method, which makes them applicable only to model-based generation approaches. The generation process and its artifacts (e.g., intermediate models) can be classified using the *Method* features. They primarily target design-time UI generation, as there are typically no intermediate artifacts in run-time generation. The *Application* features describe the running application, which consists of the generated UI (i.e., method output), its connection to the back-end and further run-time components that are (most probably) provided by the approach. The *Tools* features finally specify which tool support is available for the approach. Approaches typically do not support all features but a subset.

Most of our key features can also be seen as answers to a set of questions that will arise at some point during their development process or during the process in which an appropriate UI generation approach shall be selected to create a certain application. For these cases, we formulated key questions for the corresponding feature that also help to clarify its focus. We provide choices between discrete values only if they are clearly applicable (e.g., whether a modality is supported or not). We rely on natural language descriptions in all other cases, because a metric that is not clearly applicable will most probably lead to misunderstandings. For example, it is not intuitively clear what "high" interoperability means, neither for the person who applies the metric, nor for the person who reads the classification.

The remainder of this chapter defines each key feature in the following form:

[1] `http://ui-gen.ict.tuwien.ac.at`

- Feature Definition,
- Why this feature is important,
- Key Question(s),

for each of the five categories.

A.1 General Features

General features can be used to roughly classify UI generation approaches. Note that such approaches do not necessarily have to be model-based.

G1 – Scope

Feature Definition: This feature defines whether an approach operates at run-time, design-time or both. Run-time approaches perform all transformations exclusively at application run-time. Design-time approaches, in contrast, apply at least part of their transformations (regardless of whether they are model-to-model or model-to-text transformations) at design-time.

Moreover, it describes which aspects of the application to build can be covered by the models and the method artifacts, and which types of UIs and applications are supported by the approach.

Why this feature is important: This feature helps the application developer to get an initial impression of whether the approach will allow her to satisfy the requirements of the application to build. Information about the approach is detailed in the features to follow.

Key Question: Which aspects of an interactive application can be captured with the input models, additional artifacts and customization possibilities?

G2 – Target Devices/Platforms

Feature Definition: Basically, we rely on the platform definition of the MBUI W3C group, which says that such a platform consists of hardware and software resources. This feature distinguishes Desktop PC, Tablet PC, Smartphone or user-defined devices together with different Target Toolkits/Operating Systems (e.g., JAVA, HTML, Android or iOS). It additionally specifies whether the UI can be tailored automatically for a given device. Furthermore it clarifies whether the device supports custom widgets or the inclusion of multi-media content (e.g., audio or video data).

Why this feature is important: Applications are typically developed for a certain device/platform. This feature provides the information whether a certain device/platform is supported and additionally allows the designer to estimate the effort for adapting an application for more than one device/platform.

Key Questions: Which platform characteristics are distinguished? What target platforms (e.g., operating system, Java version) and devices (e.g., mouse or finger based) are supported? Does the approach support automated UI tailoring for different target devices?

G3 – Modalities

Feature Definition: This feature specifies which modalities are supported. Examples are graphical UIs, speech-, gesture- or other modality. This key feature also specifies which toolkits are provided for each modality, and for each toolkit, whether custom widgets are supported or not. An example of a graphical custom widget is a graphical seat selection widget for a flight-booking application.

Why this feature is important: Multi-modal interaction allows for a more "natural" interaction with a device. Such UIs may improve the usability of an application for all users or may be tailored to certain users needs (e.g., a speech UI for blind users).

Key Questions: Which modalities and toolkits are supported? To which extent does the approach support multi-media (pictures, audio and video) in- and output?

G4 – Interplay between Modalities

Feature Definition: This feature specifies the interplay between the different modalities (i.e., fusion of the modality input), based on the CARE (Complementarity, Assignment, Redundancy, and Equivalence) properties [CNS+95].

Why this feature is important: It is important for the robustness of the interaction with the final application when multiple modalities are used.

Key Question: How does the approach treat concurrent input from different modalities? Is there support beyond the CARE properties?

G5 – Compliance to Standards

Feature Definition: This feature specifies whether the approach itself, or the resulting application is compliant to any standard (e.g., ISO, DIN standards, etc. and technology standards like OMG, W3C, etc.).

Why this feature is important: This feature is important for two reasons. First, it is especially important in safety- or security-critical systems (e.g., medical or railroad applications). Second, it allows the designer to judge the interoperability of an approach (e.g., if the approach is compliant to W3C or OMG standards or uses other established languages/technologies).

G6 – Development Effort for First Prototype

Feature Definition: This key feature allows to estimate the effort needed to create a running application. It specifies which steps are necessary apart from the UI generation, to create a running application (e.g., back-end development). In case that different development paths with entry points at different levels of abstraction are supported, it specifies the highest CRF start level for fully automatic UI generation and specifies the effort for the development path that is typically used and why (e.g., minimal development effort).

Why this feature is important: Less development effort means less development time and less time-to-market. A shorter development time for the first prototype allows for early testing

and facilitates the exploration of alternatives. This feature is important as long as time-to-market and the quality of the resulting application matter.

Key Question: Which input models / artifacts (e.g., models, transformation rules, device specification, style files, other) are required? Are default values (e.g., basic transformation rules and style sheets) provided for fully automatic UI generation? Which additional steps are necessary to create a running application?

G7 – Reuse / Reusability

Feature Definition: This feature defines to which extent reuse of existing models/artifacts is supported and to which extent models/artifacts created during the development process are reusable.

Why this feature is important: This feature is especially interesting if more than one UI is created for the same application or for the creation of product lines.

Key Question: Does the approach provide a model-repository, model-patterns, transformation rules, device specification, customizations or anything else to facilitate reuse? Are (parts of the) generated applications reusable?

G8 – Development Status

Feature Definition: This feature specifies whether the approach has the maturity of a Research Prototype, an Industrial Application or at a level defined by the approach developers. It also clarifies whether the approach is still being improved or if even maintenance work was stopped.

Why this feature is important: This feature allows estimating the ease-of-use and the support the designer may expect.

G9 – Documentation

Feature Definition: This feature specifies which documentation (e.g., scientific publications, getting started or a Web-site) is available and where it can be found.

Why this feature is important: This feature is crucial as it is directly related to the ease-of-use and thus to the time needed to create an application.

Key Question: Is there any documentation at all? Which levels of expertise are supported? For example, are there examples, tutorials or getting started documents for novice users or is the code the "documentation"? Does the developer community offer support (e.g., a blog)? Are small examples, which can be used to reenact all generation and development steps of an approach available?

G10 – Terms of Use

Feature Definition: This feature clarifies conditions under which the approach is available (i.e., open source, public license, free license, license for money or other).

Why this feature is important: It is important for a designer whether she is only able to base her work on the concepts of a certain approach or if she may use the existing implementation.

Key Question: Is the implementation of the approach available in addition to the used concepts?

A.2 Input Model Features

The three features of the input model category shall be used to specify all models that are required as input for the UI generation process (i.e., method). Models that are created as intermediate artifacts during the generation process are treated in the subsequent Method chapter. If the method supports different development paths (e.g., forward and reverse engineering) with different starting points, all models are described in the current section and grouped according to starting points. These features are only applicable to model-based UI generation approaches and can be ignored if an approach is not based on explicit models (e.g., if "lightweight models" like programming APIs are used).

I1 – Name and Focus

Feature Definition: Model-based UI generation approaches typically use different models to capture different aspects of the same UI. This feature defines the name and scope (i.e., interaction-, domain-, platform-, user-, environment models or other) of a certain model together with a brief description in natural language.

Why this feature is important: This feature allows the designer to check whether all crucial requirements on the UI to create can be specified in a model and thus be taken into account by the transformation method during the development. In case that not all requirements are supported, there might still be the chance to satisfy them through the input of additional artifacts (e.g., style sheets) in the transformation method or to satisfy them through manual customization. Additional Artifacts are specified in the Method Feature M3 and support for manual customization is specified in Feature M4.

Key Question: Which aspects of the UI can be captured through a certain model?

I2 – Language

Feature Definition: This feature specifies which type of modeling languages specifies the metamodel (options are UML, Ecore, Domain-specific or other). This feature also specifies the corresponding CRF level of the model, in case that the model is a UI model.

Why this feature is important: The designer may save time and effort if she is already familiar with a certain type of model (e.g., CTT) and the corresponding tools for creating/editing them.

I3 – Input Model Scalability

Feature Definition: This feature specifies any boundaries in terms of model size, up to which models can be handled and the support for creating such models (e.g., through model patterns as basic building blocks).

Why this feature is important: This feature is important if UI generation shall be applied for building large scale (e.g., industrial) applications where small proof-of-concept implementations do not suffice.

Key Question: Are there ways to handle large scale models (e.g., through structuring them hierarchically)? Can the models or parts of them be reused, or are there templates or patterns that can be used to build new (large scale) models with a reduce effort?

A.3 Method Features

This category is dedicated to features of the transformation method (i.e., process and artifacts) of the generation approach.

M1 – Roles

Feature Definition: Model-driven UI development approaches typically distinguish different roles (e.g., interaction designer, UI designer, back-end architect, etc.). This feature specifies all required roles, the corresponding tasks of each role and which skills (i.e., modeling, programming or other) are required to perform a certain role. Note that one person can have different roles during the development of an application.

We do not distinguish levels like expert or novice for a certain role, because their interpretation is highly subjective and would most probably lead to misconceptions, but rather specify the skills that are required for a certain role (e.g., modeling, programming, etc.). This feature furthermore specifies whether a certain role relies on established techniques (e.g., UML, Cascading Style Sheets (CSS) or Java). This is once again important to lower the initial threshold and increase the ease of use/acceptance by application developers, as they might already be familiar with such techniques.

Why this feature is important: This feature specifies which skills are needed to apply a certain approach.

Key Questions: Which roles have to be filled and which skills are required for each role? Does the approach rely on established techniques and, if yes, which ones?

M2 – Development Paths / Workflow

Feature Definition: This feature defines which development paths (i.e., forward, reverse, iterative or other) [LV09] are supported by the transformation method and a standard workflow (if existing) that is supported through the transformation method. Each starting point should reference the corresponding input models as specified in Section A.2.

Why this feature is important: The support of different development paths adds flexibility and may ease the integration of an approach into an already existing workflow. Supporting a standard workflow facilitates the setup of a new "production workflow" in case that none exists.

Key Questions: Which development paths are supported by the method and what are their starting points? Is there a standard workflow for the creation of an application?

M3 – Artifacts

Feature Definition: This feature specifies artifacts that can be provided in addition to the input models (e.g., style sheets) and intermediate artifacts (e.g., models, properties files, etc.) that are created during the generation process and can be modified by the designer before the next generation step is triggered. In case that such intermediate artifacts are models, the features defined in the Input Model section (Section A.2) shall be applied. Note that intermediate artifacts are limited to design-time approaches, as run-time UI generation only supports input models/artifacts by definition.

Why this feature is important: Additional artifacts are frequently used to tailor an application for a certain device or to improve the usability of the resulting UI. Thus they are important for multi-device UI generation and customization.

M4 – Customization

Feature Definition: This feature specifies which information can be taken into account by the transformation method through additional manual input (i.e., customization), using the five sub-features presented below.

Why this feature is important: The usability of automatically generated UIs is typically rather low, because missing details are completed based on heuristics and assumptions [MPV11]. Manual customization gives the designer control over the resulting UI and is vital to improve the usability of the resulting application.

Key Question: Which customizations are supported (e.g., layout-, style-, rendering-modifications, etc.)?

As good usability is crucial for the acceptance of the application by the end user, we elaborate this feature in five sub-features to be applied for customization:

M4.1 – Accessible Models / Artifacts

This sub-feature references the specification in the M3 – Artifacts feature. In particular, it answers the following two key questions: Which artifacts are accessible for customization? Are these models/artifacts stored in a format that can be used as input to other approaches/tools?

M4.2 – Level of Expertise

This feature specifies the required skills for a certain customization (e.g., modeling, programming, etc.).

M4.3 – Support for Iterative Customization

This feature specifies if and to which extent iterative UI development is supported.

M4.4 – Support for Localization

Localization means providing the same UI with different languages. The importance of this feature depends on the targeted user group of the application, but we conjecture that localization is inevitable for a wide-spread use of an application (e.g., for smartphone application GUIs).

M4.5 – Customization Reusability

This feature specifies whether customizations performed on a specific UI of an application can be applied for other UIs for the same or similar (e.g., product-lines) applications, too. This is especially interesting if the different UIs for the same application follow the same corporate design, or for the development of product lines.

M5 – Method Scalability

Feature Definition: This feature specifies transformation method limitations in terms of scalability (e.g., maximum number of nodes for a certain model (referring to Input Model Feature I3), critical development paths, etc.). This feature needs to clarify whether a certain limitation is due to a conceptual problem or related to a particular implementation in the method's tool support.

Why this feature is important: This feature is relevant for applications that have to deal with large input models (e.g., in an industrial context). It is important as the size of the input model(s) directly influences the time required to complete the generation process.

Key Question: Are there any critical transformation steps that may prevent a successful completion of the overall process (e.g., memory overflow or excessive growth of computation time)? Which counter measures can be taken by the application developer to avoid such problems?

M6 – Adaptability

Feature Definition: This feature defines if and how a transformation method, and in particular its development paths, can be adapted.

Why this feature is important: Adapting a transformation method according to an existing workflow facilitates its integration into an existing production environment. This feature adds flexibility and broadens the context in which an approach is applicable.

Key Question: Can the transformation method be tailored to an application developer's requirements or does she have to strictly follow a (set of) given development path(s)?

A.4 Application Features

This category is dedicated to key features that specify the final running applications.

A1 – Typical Applications

Feature Definition: This feature deals with applications that have been implemented (e.g., in form of a case study), providing information about the purpose of an application, its level of maturity (e.g., research prototype or industrial application), its complexity and how much effort was spent to develop it.

Why this feature is important: The type of applications that can be generated is limited through the input models and the method. The provision of concrete applications/examples intends to illustrate an approach's capabilities.

Key Question: Is a certain approach suitable to create applications in a given domain?

A2 – Usability / User Experience

Feature Definition: This key feature reports on usability/user experience evaluations of complete applications.

Why this feature is important: A good level of usability/user experience is vital for the acceptance of the application by the end-user.

A3 – Integration of Generated UI and Back-end

Feature Definition: This feature specifies which steps are required to integrate the generated UI code with the manually created application back-end (i.e., application logic and persistence layer) and if a certain architecture is required (e.g., three tire or Model-View-Controller).

Why this feature is important: The information on state-of-the-art UI generation approaches typically ends with the resulting UI and there is scarcely any information on how to integrate the generated UI code and the application back-end. This feature facilitates the estimation of the required effort as this integration has to be completed manually in most cases.

Key Question: Does the approach support a specific type of back-end architecture or technology?

A4 – Interoperability

Feature Definition: This feature specifies if and how applications are able to interact with other devices/applications in ubiquitous computing environments.

Why this feature is important: Future environments will contain fewer dedicated computing devices (e.g., desktop PCs), but a large number of small devices with computing capabilities. The capability to connect and exploit the information and computing power of surrounding devices in an ad-hoc manner will be a vital factor for the acceptance of new devices/applications.

Key Question: Which kind of ubiquitous computing technologies are supported?

A5 – Run-time Adaptations

Feature Definition: This feature specifies whether the approach supports UI plasticity, migratory or distributed UIs. In case of design-time UI generation, it also specifies which properties (typically artifacts) of the application can be changed without re-generation in case that the transformations are applied at design-time.

Why this feature is important: This feature is important as requirements typically evolve in time, and run-time adaptations support such changes through a smooth transition that does not require re-generation and possibly not even human intervention.

Key Question: Which kind of run-time adaptations are supported?

A.5 Tool Features

This category specifies key features regarding tool support for the application developer. Tool support should hide all unnecessary details from the application developer, to lower the cognitive load and allow her to concentrate on the operation she wants to perform. A good analogy is driving a car. You do not need to now in detail how all the pieces fit together, you only have to know how to operate it [Höö00]. We conjecture that good tool support may pave the way towards UI customization by end-users. The first development, however, will always remain in the hands of experts, just like a car is not built by its driver but by engineers.

The following set of features are to be used for the specification of a certain tool:

- *Name* specifies the tool's name.
- *Model/Artifact* specifies the models/artifacts that can be created or edited with this tool, with references to the corresponding specifications provided in the Input Model specifications and in Artifact Key Feature M3.
- *Role* specifies which role, defined in key feature M1, uses the tool.
- *Type* specifies the type (i.e., graphical-, tree-editor or other) of the tool.
- *Required Skills* specifies which skills are required to use the tool. These skills can be equal or a sub-set of the skills that have been assigned to the corresponding role in the method section.
- *Maturity* specifies the implementation status of the tool (i.e., prototype, industrial or other).

The features presented above are in fact sub-features of the key features T1 to T5, defined below. They are to be applied for the models of a certain key feature.

T1 – Input Model Support

Feature Definition: This feature specifies the tool support for all models specified in Section A.2.

Why this feature is important: Strong tool support for the input model creation reduces the development time for the first running prototype and the overall development time of an application.

T2 – UI Generation Support

Feature Definition: This feature specifies the tool support for the transformation method specified in Section A.3.

Why this feature is important: Strong tool support for the transformation method may facilitate multi-device UI generation or the use of different development paths, again reducing the overall development time of an application.

T3 – Customization Support

Feature Definition: This feature specifies the tool support for manual customization.

Why this feature is important: Tool support for manual customization is important as it allows the designer to achieve a high level of usability for the resulting UIs.

T4 – Application Creation Support

Feature Definition: This feature specifies the tool support for connecting the generated UI and the application back-end.

Why this feature is important: The generated UI and the manually created back-end are typically connected manually. Tool support that may, for example, generate (parts of) the required source code to establish this connection reduces the manual programming effort, again reducing the overall application development time.

T5 – Run-time Adaptation Support

Feature Definition: This feature specifies the tool support for adaptations of the running application.

Why this feature is important: This feature is only relevant if run-time UI generation is supported by an approach. Run-time adaptations may include model inspection and changes and add flexibility to an application.

Literature

[ABB+08] Roberto Acerbis, Aldo Bongio, Marco Brambilla, Stefano Butti, Stefano Ceri, and Piero Fraternali. Web applications design and development with WebML and WebRatio 5.0. In RichardF. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 392–411. Springer Berlin Heidelberg, 2008.

[App11] Apple Inc. *Apple Human Interface Guidelines - User Experience*, May 2011.

[App12] Apple Inc. *iOS Human Interface Guidelines*, August 2012.

[AR14] David Alonso-Ríos. *Development of usability and context-of-use taxonomies, integration with techniques for the study of usability and application to real-world intelligent systems.* PhD thesis, University of A Coruña, A Coruña, Spain, to appear 2014.

[Arm14] Alexander Armbruster. A graphical editor for structural screen models. Master's thesis, Vienna University of Technology, to appear 2014.

[ARRP+14] David Alonso-Ríos, David Raneburger, Roman Popp, Hermann Kaindl, and Jürgen Falb. A user study on tailoring GUIs for smartphones. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, 2014.

[ARVGMRMB10] David Alonso-Ríos, Ana Vázquez-García, Eduardo Mosqueira-Rey, and Vicente Moret-Bonillo. Usability: A critical analysis and a taxonomy. *Int. J. Hum. Comput. Interaction*, 26(1):53–74, 2010.

[AVCF+10] Nathalie Aquino, Jean Vanderdonckt, Nelly Condori-Fernández, Óscar Dieste, and Óscar Pastor. Usability evaluation of multi-device/platform user interfaces generated by model-driven engineering. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 30:1–30:10, New York, NY, USA, 2010. ACM.

[AVPP11] Nathalie Aquino, Jean Vanderdonckt, Jose Ignacio Panach, and Oscar Pastor. Conceptual modelling of interaction. In David W. Embley and Bernhard Thalheim, editors, *Handbook of Conceptual Modeling*, pages 335–358. Springer Berlin Heidelberg, 2011.

[AVVP09] Nathalie Aquino, Jean Vanderdonckt, Francisco Valverde, and Oscar Pastor. Using profiles to support model transformations in the model-driven development of user interfaces. In Victor Lopez Jaquero, Francisco Montero Simarro, Jose Pascual Molina Masso, and Jean Vanderdonckt, editors, *Computer-Aided Design of User Interfaces VI*, pages 35–46. Springer London, 2009.

[BBC+01] A.F. Blackwell, C. Britton, A. Cox, T.R.G. Green, C. Gurr, G. Kadoda, M.S. Kutar, M. Loomes, C.L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and

R.M. Young. Cognitive dimensions of notations: Design tools for cognitive technology. In Meurig Beynon, ChrystopherL. Nehaniv, and Kerstin Dautenhahn, editors, *Cognitive Technology: Instruments of Mind*, volume 2117 of *Lecture Notes in Computer Science*, pages 325–341. Springer Berlin Heidelberg, 2001.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Morgan & Claypool, 2012.

[Bev05] Nigel Bevan. Guidelines and standards for web usability. In *Proceedings of HCI International 2005.* Lawrence Erlbaum, 2005.

[BFJ+01] George Buchanan, Sarah Farrant, Matt Jones, Harold Thimbleby, Gary Marsden, and Michael Pazzani. Improving mobile internet usability. In *Proceedings of the 10th international conference on World Wide Web*, WWW '01, pages 673–680, New York, NY, USA, 2001. ACM.

[BFS+13] Ulrike Bruckenberger, Verena Fuchsberger, Gerald Stollnberger, Barbara Weixelbaumer, Ewald Strasser, and Manfred Tscheligi. D5.3: Finaler Evaluierungsbericht (Studienergebnisse 2. NutzerInnenstudie). Technical report, Human-Computer Interaction & Usability Unit, Universität Salzburg, 2013.

[BHLV94] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, and Jean Vanderdonckt. Towards a dynamic strategy for computer-aided visual placement. In *Proceedings of the workshop on Advanced visual interfaces*, AVI '94, pages 78–87, New York, NY, USA, 1994. ACM.

[BHLV95] Francois Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, and Jean Vanderdonckt. A model-based approach to presentation: A continuum from task analysis to prototype. In Fabio Paternó, editor, *Interactive Systems: Design, Specification, and Verification*, Focus on Computer Graphics, pages 77–94. Springer Berlin Heidelberg, 1995.

[Bor00] Jan O. Borchers. A pattern approach to interaction design. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, DIS '00, pages 369–378, New York, NY, USA, 2000. ACM.

[Bot11] Goetz Botterweck. Multi front-end engineering. In Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 27–42. Springer Berlin Heidelberg, 2011.

[Bro93] Dermot Browne. *STUDIO: STructured User-Interface Design for Interaction Optimisation.* Prentice Hall, Englewood Cliffs, NJ, USA, 1993.

[Bro96] John Brooke. SUS: a "quick and dirty" usability scale". In P. W. Jordan, B. Thomas, and B. A. Weerdmeester & A. L. McClelland, editors, *Usability Evaluation in Industry*, chapter 21, pages 189–194. London: Taylor and Francis, 1996.

[CBF09] Stefano Ceri, Marco Brambilla, and Piero Fraternali. The history of WebML lessons learned from 10 years of model-driven development of Web applications. In Alexander T. Borgida, Vinay K. Chaudhri, Paolo Giorgini, and Eric S. Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 273–292. Springer Berlin Heidelberg, 2009.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent

Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[CFK+04] Adrien Coyette, Stéphane Faulkner, Manuel Kolp, Quentin Limbourg, and Jean Vanderdonckt. SketchiXML: towards a multi-agent design tool for sketching user interfaces based on UsiXML. In *Proceedings of the 3rd annual conference on Task models and diagrams*, TAMODIA '04, pages 75–82, New York, NY, USA, 2004. ACM.

[CLV+03] Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Bergh, and Bert Creemers. Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. In *Human-Computer Interaction with Mobile Devices and Services*, volume 2795 of *Lecture Notes in Computer Science*, pages 256–270. Springer Berlin / Heidelberg, 2003.

[CNS+95] Joëlle Coutaz, Laurence Nigay, Daniel Salber, Ann Blandford, Jon May, and Richard M. Young. Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. In *Proceedings of INTERACT 95*, pages 115–120, 1995.

[CP09] Li Chen and Pearl Pu. Interaction design guidelines on critiquing-based recommender systems. *User Modeling and User-Adapted Interaction*, 19(3):167–206, 2009.

[ELP11] David W. Embley, Stephen W. Liddle, and Oscar Pastor. Conceptual-model programming: A manifesto. In David W. Embley and Bernhard Thalheim, editors, *Handbook of Conceptual Modeling*, pages 3–16. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-15865-0_1.

[EPV09] Dominik Ertl, Roman Popp, and Matthieu Vallee. Key disciplines in multimodal user interface development with interaction design. In *Proceedings of the Fourth IASTED International Conference on Human-Computer Interaction (IASTED-HCI 2009)*, 2009.

[Ert11] Domink Ertl. *Semi-Automatic Generation of Multimodal User Interfaces for Dialogue-based Interactive Systems*. PhD thesis, E384, 2011.

[FBK+08] Sebastian Feuerstack, Marco Blumendorf, Maximilian Kern, Michael Kruppa, Michael Quade, Mathias Runge, and Sahin Albayrak. Automated usability evaluation during model-based interactive system development. In *Proceedings of the 2nd Conference on Human-Centered Software Engineering and 7th International Workshop on Task Models and Diagrams*, HCSE-TAMODIA '08, pages 134–141, Berlin, Heidelberg, 2008. Springer-Verlag.

[FBSA08] Sebastian Feuerstack, Marco Blumendorf, Veit Schwartze, and Sahin Albayrak. Model-based layout generation. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '08, pages 217–224, New York, NY, USA, 2008. ACM.

[FCDC10] Alfonso García Frey, Gaëlle Calvary, and Sophie Dupuy-Chessa. Self-explanatory user interfaces by model-driven engineering. In *Proceedings of the second ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*, pages 341–344. ACM Press, 2010.

[FKH+06] Jürgen Falb, Hermann Kaindl, Helmut Horacek, Cristian Bogdan, Roman Popp, and Edin Arnautovic. A discourse model for interaction design based on theories of human communication. In *Extended Abstracts on Human Factors in Computing Systems (CHI '06)*, pages 754–759. ACM Press: New York, NY, 2006.

[FKP+09] Jürgen Falb, Sevan Kavaldjian, Roman Popp, David Raneburger, Edin Arnautovic, and Hermann Kaindl. Fully automatic user interface generation from discourse models. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '09)*, pages 475–476. ACM Press: New York, NY, 2009.

[FPR+07] Jürgen Falb, Roman Popp, Thomas Röck, Helmut Jelinek, Edin Arnautovic, and Hermann Kaindl. UI prototyping for multiple devices through specifying interaction design. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT 2007)*, pages 136–149, Rio de Janeiro, Brazil, September 2007. Springer.

[GFCDC+12] Alfonso García Frey, Eric Céret, Sophie Dupuy-Chessa, Gaëlle Calvary, and Yoann Gabillon. UsiComp: an extensible model-driven composer. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 263–268, New York, NY, USA, 2012. ACM.

[GFCDCM13] Alfonso García-Frey, Gaëlle Calvary, Sophie Dupuy-Chessa, and Nadine Mandran. Model-based self-explanatory UIs for free, but are they valuable? In *Proceedings of the 14th IFIP TC13 Conference on Human-Computer Interaction (INTERACT'13), 2-6 September 2013, Cape Town, South Africa*, pages 144–161. Springer, 2013.

[GW04] Krzysztof Gajos and Daniel S. Weld. SUPPLE: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interface (IUI '04)*, pages 93–100, New York, NY, USA, 2004. ACM Press.

[GW05] Krzysztof Gajos and Daniel S. Weld. Preference elicitation for interface optimization. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, UIST '05, pages 173–182, New York, NY, USA, 2005. ACM.

[GWW08] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. Decision-theoretic user interface generation. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 2008.

[GWW10] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. Automatically generating personalized user interfaces with supple. *Artificial Intelligence*, 174(12 - 13):910 – 950, 2010.

[HME11] Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke (Eds.), editors. *Model-Driven Development of Advanced User Interfaces*, volume Vol. 340 of *Studies in Computational Intelligence*. Springer-Verlag Berlin Heidelberg, 2011.

[Höö00] K. Höök. Steps to take before intelligent user interfaces become real. *Interacting with Computers*, 12(4):409 – 426, 2000.

[HPR12] Helmut Horacek, Roman Popp, and David Raneburger. Automated generation of user interfaces - a comparison of models and future prospects. In Maurtua Inaki, editor, *Human Machine Interaction - Getting Closer*. InTech, 2012.

[HSW12] Frank Honold, Felix Schüssel, and Michael Weber. Adaptive probabilistic fission for multimodal systems. In *OzCHI'12 – Proceedings of the ACM OzCHI 2012*, Melbourne, Australia, November, 26–30 2012. ACM.

[IFM13] Interaction Flow Modeling Language (IFML), 03 2013.

[JJM+05] Steve Jones, Matt Jones, Gary Marsden, Dynal Patel, and Andy Cockburn. An evaluation of integrated zooming and scrolling on small screens. *Int. J. Hum.-Comput. Stud.*, 63(3):271–303, September 2005.

[JMMN+99] Matt Jones, Gary Marsden, Norliza Mohd-Nasir, Kevin Boone, and George Buchanan. Improving Web interaction on small displays. *Comput. Netw.*, 31(11-16):1129–1137, May 1999.

[Kai89] Hermann Kaindl. *Problemlösen durch heuristische Suche in der Artificial Intelligence*. Springer-Verlag, 1989.

[Kav11] Sevan Kavaldjian. *Automated Model-driven Generation of the Structure of WIMP User Interfaces Based on High-level Models*. PhD thesis, Vienna University of Technology, 2011.

[KB13] F. Kis and C. Bogdan. Lightweight low-level query-centric user interface modeling. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 440–449, 2013.

[KBFK08] Sevan Kavaldjian, Cristian Bogdan, Jürgen Falb, and Hermann Kaindl. Transforming discourse models to structural user interface models. In *Models in Software Engineering, LNCS 5002*, volume 5002/2008, pages 77–88. Springer, Berlin / Heidelberg, 2008.

[KF90] Won Chul Kim and James D. Foley. DON: user interface presentation design assistant. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, UIST '90, pages 10–20, New York, NY, USA, 1990. ACM.

[KFK09a] Sevan Kavaldjian, Jürgen Falb, and Hermann Kaindl. Fully automatic content presentation specific to intentions. In *Proceedings of the IUI'09 Workshop on Model Driven Development of Advanced User Interfaces*, 2009.

[KFK09b] Sevan Kavaldjian, Jürgen Falb, and Hermann Kaindl. Generating content presentation according to purpose. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA, Oct. 2009.

[Kit04] Barbara Kitchenham. Procedures for performing systematic reviews. Technical report, Software Engineering Group, Department of Computer Science, Keele University, Keele, Staffs ST5 5BG, UK and Empirical Software Engineering, National ICT Australia Ltd., Bay 15 Locomotive Workshop Australian Technology Park, Garden Street, Eversleigh, NSW 1430, Australia, 2004.

[KJ02] Hermann Kaindl and Rudolf Jezek. From usage scenarios to user interface elements in a few steps. In Christophe Kolski and Jean Vanderdonckt, editors, *Proceedings of CADUI'02*, pages 91–102. Kluwer, 2002.

[KL10] Richard Kennard and John Leaney. Towards a general purpose architecture for UI generation. *Journal of Systems and Software*, 83(10):1896–1906, October 2010.

[KP02] Heikki Keränen and Johan Plomp. Adaptive runtime layout of hierarchical UI components. In *Proceedings of the second Nordic conference on Human-computer interaction*, NordiCHI '02, pages 251–254, New York, NY, USA, 2002. ACM.

[KPR12] Hermann Kaindl, Roman Popp, and David Raneburger. Automated generation of user interfaces: Based on use case or interaction design specifications? In Slimane Hammoudi, Marten van Sinderen, and José Cordeiro, editors, *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT'12)*, pages 303–308. SciTePress, July 2012.

[KR03] Anne Kaikkonen and Virpi Roto. Navigating in a mobile XHTML application. In *Proceedings of the SIGCHI Conference on Human Factors in Computing*

Systems, CHI '03, pages 329–336, New York, NY, USA, 2003. ACM.

[KRF+09] Sevan Kavaldjian, David Raneburger, Jürgen Falb, Hermann Kaindl, and Dominik Ertl. Semi-automatic user interface generation considering pointing granularity. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA, Oct. 2009.

[LCCV03] Kris Luyten, Tim Clerckx, Karin Coninx, and Jean Vanderdonckt. Derivation of a dialog model from a task model by activity chain extraction. In *Interactive Systems. Design, Specification, and Verification*, volume 2844 of *Lecture Notes in Computer Science*, pages 83–83. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39929-2_14.

[LD10] Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer Berlin Heidelberg, 2010.

[Lei10] Michael Leitner. Space-saving placement using a structural user interface model. Master's thesis, Vienna University of Technology (TU-Wien), Faculty of Electrical Engineering and Information Technology, Institute of Computer Technology, E384, 2010.

[LF01] Simon Lok and Steven Feiner. A survey of automated layout techniques for information presentations, 2001.

[LF02] Simon Lok and Steven K. Feiner. The AIL automated interface layout system. In *Proceedings of the 7th international conference on Intelligent user interfaces*, IUI '02, pages 202–203, New York, NY, USA, 2002. ACM.

[LFG90] Paul Luff, David Frohlich, and Nigel Gilbert. *Computers and Conversation*. Academic Press, London, UK, January 1990.

[LFN04] Simon Lok, Steven Feiner, and Gary Ngai. Evaluation of visual balance for automated layout. In *Proceedings of the 9th international conference on intelligent user interfaces*, IUI '04, pages 101–108, New York, NY, USA, 2004. ACM.

[LV03] Q. Limbourg and Jean Vanderdonckt. Comparing task models for user interface design. In D. Diaper and N. Stanton, editors, *The Handbook of Task Analysis for Human-Computer Interaction*, chapter 6. Lawrence Erlbaum Associates, Mahwah, NJ, USA, 2003.

[LV09] Quentin Limbourg and Jean Vanderdonckt. Multipath transformational development of user interfaces with graph transformations. In Ahmed Seffah, Jean Vanderdonckt, and Michel C. Desmarais, editors, *Human-Centered Software Engineering*, Human-Computer Interaction Series, pages 107–138. Springer London, 2009. 10.1007/978-1-84800-907-3_6.

[LW97] Clayton Lewis and Cathleen Wharton. Cognitive walkthroughs. In Prasad V. Prabhu Martin G. Helander, Thomas K. Landauer, editor, *Handbook of Human-Computer Interactions*. Elsevier Press, Amsterdam, completely revised 2nd edition edition, 1997.

[MBS11] Gerrit Meixner, Kai Breiner, and Marc Seissler. *Model-Driven Useware Engineering*, chapter 1, pages 1–26. Studies in Computational Intelligence, SCI. Springer, Heidelberg, January 2011.

[MCC13] Gerrit Meixner, Gaëlle Calvary, and Joëlle Coutaz. Introduction to model-

based user interfaces. Technical report, The World Wide Web Consortium (W3C), 2013.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7:3–28, March 2000.

[Mic10] Microsoft Corporation. *User Experience and Interaction Guidelines for Windows 7 and Windows Vista*, Sept. 2010.

[Mic12] Microsoft Corporation. *Windows 8 User Experience Guidelines*, Aug. 2012.

[MM03] Eds.: Joaquin Miller and Jishnu Mukerjij. MDA guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[MMG+13] Christopher Mayer, Martin Morandell, Matthias Gira, Miroslav Sili, Martin Petzold, Sascha Fagel, Christian Schüller, Jan Bobeth, and Susanne Schmehl. User interfaces for older adults. In Constantine Stephanidis and Margherita Antona, editors, *Universal Access in Human-Computer Interaction. User and Context Diversity*, volume 8010 of *Lecture Notes in Computer Science*, pages 142–150. Springer Berlin Heidelberg, 2013.

[MP11] Marco Manca and Fabio Paternò. Distributing user interfaces with MARIA. In *Proceedings of the CHI'11 Workshop on Distributed User Interfaces (DUI 2011)*, 2011.

[MPS04] G. Mori, F. Paternò, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering*, 30(8):507–520, 8 2004.

[MPV11] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, present, and future of model-based user interface development. *i-com*, 10(3):2–10, November 2011.

[MR92] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM.

[MSO12] Gerrit Meixner, Marc Seissler, and Marius Orfgen. Specification and application of a taxonomy for task models in model-based user interface development environments. *International Journal on Advances in Intelligent Systems*, 4(3+4):388–398, 4 2012.

[MT88] W. C. Mann and S.A. Thompson. Rhetorical Structure Theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988.

[MT90] Silvano Martello and Paolo Toth. *Knapsack Problems – Algorithms and Computer Implementations.* John Wiley & Sons, 1990.

[MV08] Benjamin Michotte and Jean Vanderdonckt. GrafiXML, a multi-target user interface builder based on UsiXML. In *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, pages 15–22, Washington, DC, USA, 2008. IEEE Computer Society.

[MVLC08] Jan Meskens, Jo Vermeulen, Kris Luyten, and Karin Coninx. Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '08, pages 233–240, New York, NY, USA, 2008. ACM.

[Nas12] Victor Nassar. Common criteria for usability review. *Work: A Journal of Prevention, Assessment and Rehabilitation*, 41:1053–1057, 2012.

[Nie93] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[Nie94] Jakob Nielsen. Usability inspection methods. In *Conference Companion on Human Factors in Computing Systems*, CHI '94, pages 413–414, New York, NY, USA, 1994. ACM.

[OP07] Juan Carlos Molina Oscar Pastor. *Model-Driven Architecture in Practice*. Springer Berlin, 2007.

[PBM+11] Philippe Palanque, Eric Barboni, Célia Martinie, David Navarre, and Marco Winckler. A model-based approach for supporting engineering usability evaluation of interaction techniques. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 21–30, New York, NY, USA, 2011. ACM.

[PBSK99] Fabio Paternò, Ilse M. Breedvelt-Schouten, and Nicole de Koning. Deriving presentations from task models. In *Proceedings of the IFIP TC2/TC13 WG2.7/WG13.4 Seventh Working Conference on Engineering for Human-Computer Interaction*, pages 319–337, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[PEPA08] Oscar Pastor, Sergio España, José Ignacio Panach, and Nathalie Aquino. Model-driven development. *Informatik Spektrum*, 31(5):394–407, 2008.

[PFA+09] Roman Popp, Jürgen Falb, Edin Arnautovic, Hermann Kaindl, Sevan Kavaldjian, Dominik Ertl, Helmut Horacek, and Cristian Bogdan. Automatic generation of the behavior of a user interface from a high-level discourse model. In *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42)*, Piscataway, NJ, USA, 2009. IEEE Computer Society Press.

[PFRK12] Roman Popp, Jürgen Falb, David Raneburger, and Hermann Kaindl. A transformation engine for model-driven UI generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 281–286, New York, NY, USA, 2012. ACM.

[PGIP01] Oscar Pastor, Jaime Gómez, Emilio Insfrán, and Vicente Pelechano. The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Information Systems*, 26(7):507 – 534, 2001.

[PH11] Andreas Pleuss and Heinrich Hussmann. Model-driven development of interactive multimedia applications with MML. In Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 199–218. Springer Berlin Heidelberg, 2011.

[PHDB12] Andreas Pleuss, Benedikt Hauptmann, Deepak Dhungana, and Goetz Botterweck. User interface engineering for software product lines: the dilemma between automation and usability. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering Interactive Computing Systems*, EICS '12, pages 25–34, New York, NY, USA, 2012. ACM.

[PHKB12] Andreas Pleuss, Benedikt Hauptmann, Markus Keunecke, and Goetz Botterweck. A case study on variability in user interfaces. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 6–10, New York, NY, USA, 2012. ACM.

[PKR13] Roman Popp, Hermann Kaindl, and David Raneburger. Connecting interaction models and application logic for model-driven generation of Web-based graphical user interfaces. In *Proceedings of the 20th Asia-Pacific Software*

Engineering Conference (APSEC 2013), 2013.

[PM07] Oscar Pastor and Juan Carlos Molina. Presentation model. In *Model-Driven Architecture in Practice*, pages 147–189. Springer Berlin Heidelberg, 2007. 10.1007/978-3-540-71868-0_10.

[PMM97] Fabio Paternò, Cristian Mancini, and Silvia Meniconi. ConcurTaskTrees: A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 Sixth International Conference on Human-Computer Interaction*, pages 362–369, 1997.

[Pop09] Roman Popp. Defining Communication in SOA Based on Discourse Models. In *Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*, pages 829–830. ACM Press: New York, NY, 2009.

[Pop12] Roman Popp. A unified solution for service-oriented architecture and user interface generation through discourse-based communication models. Doctoral dissertation, Vienna University of Technology, Vienna, Austria, 2012.

[PR11] Roman Popp and David Raneburger. A High-Level Agent Interaction Protocol Based on a Communication Ontology. In Christian Huemer, Thomas Setzer, Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, and Clemens Szyperski, editors, *E-Commerce and Web Technologies*, volume 85 of *Lecture Notes in Business Information Processing*, pages 233–245. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-23014-1_20.

[PRK13] Roman Popp, David Raneburger, and Hermann Kaindl. Tool support for automated multi-device GUI generation from discourse-based communication models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, New York, NY, USA, 2013. ACM.

[PRS11] Jenny Preece, Yvonne Rogers, and Helen Sharp. *Interaction design: beyond human-computer interaction*. John Wiley & Sons, 3rd edition, 2011.

[PS02] Fabio Paternò and Carmen Santoro. One model, many interfaces. In *Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces (CADUI 2002)*, pages 143–154, 2002.

[PS12] Fabio Paternò and Carmen Santoro. A logical framework for multi-device user interfaces. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 45–50, New York, NY, USA, 2012. ACM.

[PSS09a] Fabio Paternò, Carmen Santoro, and Lucio Spano. Model-based design of multi-device interactive applications based on web services. In Tom Gross, Jan Gulliksen, Paula Kotzé, Lars Oestreicher, Philippe Palanque, Raquel Prates, and Marco Winckler, editors, *Human-Computer Interaction - INTERACT 2009*, volume 5726 of *Lecture Notes in Computer Science*, pages 892–905. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03655-2_98.

[PSS09b] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16:19:1–19:30, November 2009.

[PSS10a] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Exploiting Web service annotations in model-based user interface development. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, pages 219–224, New York, NY, USA, 2010. ACM.

[PSS10b] Fabio Paternó, Carmen Santoro, and Lucio Davide Spano. User task-based development of multi-device service-oriented applications. In *Proceedings of the International Conference on Advanced Visual Interfaces*, AVI '10, pages 407–407, New York, NY, USA, 2010. ACM.

[Pue97] A.R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4):40–47, 1997.

[PVE+07] Inés Pederiva, Jean Vanderdonckt, Sergio España, Ignacio Panach, and Oscar Pastor. The beautification process in model-driven engineering of user interfaces. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction — INTERACT 2007, Part I, LNCS 4662*, pages 411–425, Rio de Janeiro, Brazil, Sept. 2007. Springer Berlin / Heidelberg.

[PWB13] Andreas Pleuss, Stefan Wollny, and Goetz Botterweck. Model-driven development and evolution of customized user interfaces. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, New York, NY, USA, 2013. ACM.

[PZ10] Fabio Paternò and Giuseppe Zichittella. Desktop-to-mobile web adaptation through customizable two-dimensional semantic redesign. In Regina Bernhaupt, Peter Forbrig, Jan Gulliksen, and Marta Lárusdóttir, editors, *Human-Centred Software Engineering*, volume 6409 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16488-0_7.

[Ran08] David Raneburger. Automated graphical user interface generation based on an abstract user interface specification. Master's thesis, Vienna University of Technology, Vienna, Austria, 2008.

[Ran10] David Raneburger. Interactive model driven graphical user interface generation. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*, pages 321–324, New York, NY, USA, 2010. ACM.

[RARP+13] David Raneburger, David Alonso-Ríos, Roman Popp, Hermann Kaindl, and Jürgen Falb. A user study with GUIs tailored for smartphones. In Paula Kotzé, Gary Marsden, Gitte Lindgaard, Janet Wesson, and Marco Winckler, editors, *Human-Computer Interaction – INTERACT 2013*, volume 8118 of *Lecture Notes in Computer Science*, pages 505–512. Springer, 2013.

[RKP+14] David Raneburger, Hermann Kaindl, Roman Popp, Vedran Šajatović, and Alexander Armbruster. A process for facilitating interaction design through automated GUI generation. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, 2014.

[RLS+11] Dirk Roscher, Grzegorz Lehmann, Veit Schwartze, Marco Blumendorf, and Sahin Albayrak. Dynamic distribution and layouting of model-based user interfaces in smart environments. In Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 171–197. Springer Berlin Heidelberg, 2011.

[RMB13] David Raneburger, Gerrit Meixner, and Marco Brambilla. Platform-independence in model-based multi-device UI development. In *Proceedings of the 8th International Joint Conference on Software Technologies (ICSOFT'13)*. SciTePress, July 2013.

[RPAR+13] David Raneburger, Roman Popp, David Alonso-Ríos, Hermann Kaindl, and

Jürgen Falb. A user study with GUIs tailored for smartphones and tablet PCs. In *Proceedings of the 2013 IEEE International Conference on Systems, Man and Cybernetics (SMC 2013)*, Manchester, UK, Oct. 2013.

[RPK+11a] David Raneburger, Roman Popp, Hermann Kaindl, Jürgen Falb, and Dominik Ertl. Automated Generation of Device-Specific WIMP UIs: Weaving of Structural and Behavioral Models. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 41–46, New York, NY, USA, 2011. ACM.

[RPK+11b] David Raneburger, Roman Popp, Sevan Kavaldjian, Hermann Kaindl, and Jürgen Falb. Optimized GUI generation for small screens. In Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 107–122. Springer Berlin / Heidelberg, 2011.

[RPK13a] David Raneburger, Roman Popp, and Hermann Kaindl. Design alternatives for GUI development with discourse-based communication models. In Matthias Horbach, editor, *Informatik angepasst an Mensch, Organisation und Umwelt*, volume P-220. Gesellschaft für Informatik, Lecture Notes in Informatics (LNI), 2013.

[RPK13b] David Raneburger, Roman Popp, and Hermann Kaindl. Model-driven transformation for optimizing PSMs: A case study of rule design for multi-device GUI generation. In *Proceedings of the 8th International Joint Conference on Software Technologies (ICSOFT'13)*. SciTePress, July 2013.

[RPK+14] David Raneburger, Roman Popp, Hermann Kaindl, Alexander Armbruster, and Vedran Šajatović. An iterative and incremental process for interaction design through automated GUI generation. In *Proceedings of the 16th International Conference on Human-Computer Interaction*, 2014.

[RPKF11] David Raneburger, Roman Popp, Hermann Kaindl, and Jürgen Falb. Automated WIMP-UI behavior generation: Parallelism and granularity of communication units. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, pages 2816 –2821, Oct. 2011.

[RPV12] David Raneburger, Roman Popp, and Jean Vanderdonckt. An automated layout approach for model-driven WIMP-UI generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[RSKF11] David Raneburger, Alexander Schörkhuber, Hermann Kaindl, and Jürgen Falb. UI Development Support through Model-integrity Checks in a Discourse-based Generation Framework. In *Proceedings of the First International Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeisMoto)*, 2011.

[RWP+13] David Raneburger, Barbara Weixelbaumer, Roman Popp, Jürgen Falb, Nicole Mirnig, Astrid Weiss, Manfred Tscheligi, and Brigitte Ratzer. A case study in automated GUI generation for multiple devices. In *Proceedings of the 11th IEEE AFRICON Conference*, pages 1212–1217, 2013.

[SBD+10] Marc Seißler, K. Breiner, P. Diebold, C. Wiehr, and Gerrit Meixner. Smartmote - ein HCI-Pattern-basiertes universelles Bediengerät für intelligente Produktionsumgebungen. In *Proceedings of USEWARE 2010*, VDI-Berichte/VDI-Tagungsbände. VDI Wissensforum, 10 2010.

[Sch00] Thomas Schmid. *Untersuchung von Styleguides fü graphische Bedien-*

schnittstellen. PhD thesis, Technische Universtität Wien, 2000.

[Sch06] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[Sch10] Alexander Schörkhuber. Integritätsprüfung von Diskursmodellen, Transformationsregeln und strukturellen Modellen von graphischen User Interfaces. Master's thesis, Technische Universität Wien, Fakultät für Elektrotechnik und Informationstechnik, Institut für Computertechnik, E384, 2010.

[Sea69] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, England, 1969.

[Sea93] Andrew Sears. Layout appropriateness: A metric for evaluating user interface widget layout. *IEEE Transactions on Software Engineering*, 19:707–719, 1993.

[SGR+05] Daniel Sinnig, Ashraf Gaffar, Daniel Reichart, Peter Forbrig, and Ahmed Seffah. Patterns in model-based engineering. In Robert Jacob, Quentin Limbourg, and Jean Vanderdonckt, editors, *Computer-Aided Design of User Interfaces IV*, pages 197–210. Springer Netherlands, 2005. 10.1007/1-4020-3304-4_16.

[SK03] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42 – 45, sept.-oct. 2003.

[SMV07] Kênia Sousa, Hildeberto Mendonça, and Jean Vanderdonckt. Towards method engineering of model-driven user interface development. In *Proceedings of the 6th international conference on Task models and diagrams for user interface design*, TAMODIA'07, pages 112–125, Berlin, Heidelberg, 2007. Springer-Verlag.

[SPAS97] A. Savidis, A. Paramythis, D. Akoumianakis, and C. Stephanidis. Designing user-adapted interfaces: The unified design method for transformable interactions. In *Proceedings of the Conference on Designing Interactive Systems (DIS'97)*, pages 323–334, New York, NY, USA, 1997. ACM Press.

[SS02] Kirsten Swearingen and Rashmi Sinha. Interaction design for recommender systems. In *Designing Interactive Systems*. ACM Press, 2002.

[SST09] Grischa Schmiedl, Markus Seidl, and Klaus Temper. Mobile phone web browsing: a study on usage and usability of the mobile web. In *Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '09, pages 70:1–70:2, New York, NY, USA, 2009. ACM.

[Sze96] Pedro Szekely. Retrospective and challenges for model-based interface development. In *Design, Specification and Verification of Interactive Systems '96*, pages 1–27. Springer-Verlag, 1996.

[Tru06] Frank Truyen. The Fast Guide to Model Driven Architecture - The basics of Model Driven Architecture, January 2006.

[Van05] Jean Vanderdonckt. A MDA-compliant environment for developing user interfaces of information systems. In *CAiSE*, pages 16–31, 2005.

[Van08] Jean M. Vanderdonckt. Model-driven engineering of user interfaces: Promises, successes, and failures. In *Proceedings of 5th Annual Romanian Conf. on Human-Computer Interaction*, pages 1–10. Matrix ROM, Sept. 2008.

[vC07] Jan van den Bergh and Karin Coninx. From task to dialog model in the UML. In *Proceedings of the 6th International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA 2007), LNCS 4849*, pages 98–111, Toulouse, France, Nov 2007. Springer.

[VG94] Jean Vanderdonckt and Xavier Gillo. Visual techniques for traditional and multimedia layouts. In *Proceedings of the workshop on Advanced visual interfaces*, AVI '94, pages 95–104, New York, NY, USA, 1994. ACM.

[Š14] Vedran Šajatović. Improved tool support for model-driven development of interactive applications in UCP. Master's thesis, Vienna University of Technology, 2014.

[vSvdVB97] Mark van Setten, Gerrit C. van der Veer, and Sjaak Brinkkemper. Comparing interaction design techniques: A method for objective comparison to find the conceptual basis for interaction design. In *Proceedings of the Conference on Designing Interactive Systems (DIS'97)*, pages 349–357, New York, NY, USA, 1997. ACM Press.

[WdMFP11] Willian Massami Watanabe, Renata Pontin de Mattos Fortes, and Maria da Graça Campos Pimentel. The link-offset-scale mechanism for improving the usability of touch screen displays on the web. In *Proceedings of the 13th IFIP TC 13 international conference on Human-computer interaction - Volume Part III*, INTERACT'11, pages 356–372, Berlin, Heidelberg, 2011. Springer-Verlag.

[WF09] Andreas Wolff and Peter Forbig. Deriving user interfaces from task models. In *Proceedings of the IUI'09 Workshop on Model Driven Development of Advanced User Interfaces*, 2009.

[WF10] Andreas Wolff and Peter Forbig. Model-driven user interface development with the eclipse modeling project. In *Proceedings of the 5th International Workshop on Model Driven Development of Adavanced User Interfaces (MDDAUI 2010): Bridging between User Experience and UI Engineering*, 2010.

[WMFT13] Barbara Weixelbaumer, Nicole Mirnig, Verena Fuchsberger, and Manfred Tscheligi. D5.2: Bericht über die Ergebnisse des Cognitive Walkthrough. Technical report, Human-Computer Interaction & Usability Unit, Universität Salzburg, 2013.

[WMW+12] B. Weixelbaumer, N. Mirnig, A. Weiss, B. Ratzer, J. Falb, R. Popp, and D. Raneburger. Gender-inclusive user interface guidelines. Technical report, University of Salzburg and Vienna University of Technology, 2012.

[WMWT12] Barbara Weixelbaumer, Nicole Mirnig, Astrid Weiss, and Manfred Tscheligi. D2.2: Bericht über Studienergebnisse. Technical report, Human-Computer Interaction & Usability Unit, Universität Salzburg, 2012.

[WVDSD10] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *Software and Systems Modeling*, 9:285–309, 2010. 10.1007/s10270-009-0134-3.